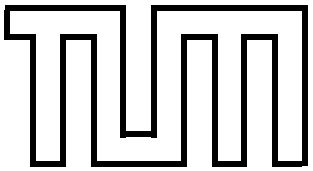


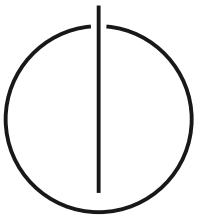
FAKULTÄT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN

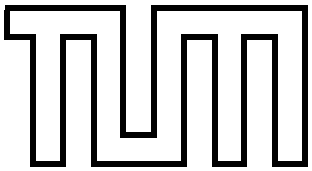
LEHRSTUHL FÜR NETZARCHITEKTUREN UND NETZDIENSTE

Master's Thesis in Informatics

DECENTRALIZED EVALUATION OF REGULAR EXPRESSIONS FOR CAPABILITY DISCOVERY IN PEER-TO-PEER NETWORKS

Maximilian Szengel





FAKULTÄT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN

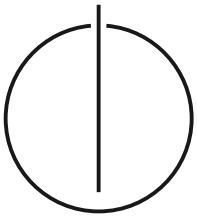
LEHRSTUHL FÜR NETZARCHITEKTUREN UND NETZDIENSTE

Master's Thesis in Informatics

DECENTRALIZED EVALUATION OF REGULAR EXPRESSIONS FOR CAPABILITY DISCOVERY IN PEER-TO-PEER NETWORKS

VERTEILTE AUSWERTUNG VON REGULÄREN AUSDRÜCKEN ZUR CAPABILITY DISCOVERY IN PEER-TO-PEER NETZWERKEN

AUTHOR: Maximilian Szengel
SUPERVISOR: Christian Grothoff, PhD (UCLA)
ADVISORS: Ralph Holz
Bartlomiej Polot
Dr. Heiko Niedermayer
DATE: November 15th, 2012



DECLARATION

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, November 15th, 2012

Maximilian Szengel

ACKNOWLEDGMENTS

I would like to thank my advisors for their guidance, extensive support and advice during the course of this work. I thank Christian Grothoff for providing the initial idea and the possibility of working with him, as well as his outstanding support on all aspects of this thesis. I thank Ralph Holz for bringing us all together on this topic and for his invaluable discussions. I thank Heiko Niedermayer for his helpful feedback and discussions on the design and related work. Moreover I thank Bartlomiej Polot for his work on the mesh part and his support during the evaluation. Furthermore, I would like to thank David Barksdale for his discussions on the design and I would like to thank Sree Harsha Totakura for his work on the GNUnet testbed and support on running the emulation.

Finally I would like to thank my friends and family for supporting me during my whole academic studies.

ABSTRACT

This thesis presents a novel approach for decentralized evaluation of regular expressions for capability discovery in Distributed Hash Table (DHT)-based overlays. The system provides support for announcing capabilities expressed as regular expressions and discovering participants offering adequate capabilities.

The idea behind our approach is to convert regular expressions into finite automatons and store the corresponding states and transitions in a DHT. We show how locally constructed Deterministic Finite Automatons (DFAs) are merged in the DHT into an Nondeterministic Finite Automaton (NFA) without the knowledge of any NFA already present in the DHT and without the need for any central authority. Furthermore we present options of optimizing the DFAs.

There exist several possible applications for this general approach of decentralized regular expression evaluation. However, in this thesis we focus on the application of discovering users that are willing to provide network access using a specified protocol to a particular destination.

We have implemented the system for our proposed approach and conducted a simulation. Moreover we present the results of an emulation of the implemented system in a cluster.

ZUSAMMENFASSUNG

Diese Arbeit präsentiert einen neuartigen Ansatz zum dezentralisierten Auswerten von regulären Ausdrücken, zur Capability Discovery in Distributed Hash Table (DHT) basierten overlay Netzwerken. Das System ermöglicht das bekanntmachen von Capabilities, welche durch reguläre Ausdrücke beschrieben werden und das Auffinden von Teilnehmern mit dazu passenden Capabilities.

Die Idee hinter unserem Ansatz ist es reguläre Ausdrücke in finite Automaten umzuwandeln und die dazugehörigen Knoten und Transitionen in einer DHT abzulegen. Wir zeigen wie lokal erstellte Deterministic Finite Automatons (DFAs) in der DHT zu einem Nondeterministic Finite Automaton (NFA) fusioniert werden ohne benötigte Kenntnis über möglicherweise schon vorhandene NFAs in der DHT, sowie ohne Notwendigkeit für eine zentrale Authority. Weiterhin präsentieren wir optionen zum optimieren der DFAs.

Für diesen Ansatz gibt es viele mögliche Anwendungsmöglichkeiten, allerdings konzentrieren wir uns in dieser Arbeit auf das Auffinden von Usern, die bereit sind Netzwerkzugriff für ein spezielles Protokoll zu einer bestimmten Destination bereitzustellen.

Wir haben das System für unseren vorgestellten Ansatz implementiert und simuliert. Weiterhin präsentieren wir die Ergebnisse einer Emulation des Systems mit Hilfe eines Clusters.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

GLOSSARY

R^5N	Randomized Recursive Routing for Restricted-Route Networks. 40
API	Application Programming Interface. 4, 39, 40, 42, 43, 45
AS	Autonomous System. 46, 51, 52, 73
CAIDA	The Cooperative Association for Internet Data Analysis. 46
CCDF	Complementary Cumulative Distribution Function. 49
CISS	Cooperative Information Sharing System. 63, 64
DFA	Deterministic Finite Automaton. ix, xi, 3, 5–8, 10, 12–19, 21–34, 36–38, 40–43, 46, 47, 51–53, 55, 66
DHT	Distributed Hash Table. ix, xi, 1, 3, 4, 24, 26, 27, 29, 30, 32–42, 47, 48, 53, 54, 56–59, 62–64, 66
DNS	Domain Name System. 44
DPI	Deep Packet Inspection. 16
DPMS	Distributed Pattern Matching System. 61, 62
FSM	Finite-State Machine. 5
GNUnet	GNU's Framework for Secure Peer-to-Peer Networking. 39–41, 44, 45, 51, 54
IPv4	Internet Protocol version 4. 9, 10, 37, 38, 42–47, 49, 51, 52, 64
IPv6	Internet Protocol version 6. 42–45
NFA	Nondeterministic Finite Automaton. ix, xi, 1, 5–8, 10–14, 19, 25–38, 41–43, 46–51, 55
P2P	Peer-to-Peer. 1, 3, 39, 47, 56, 60, 62, 64

pfx2as	Routeviews Prefix to AS mappings Dataset. 46, 47, 51
PHT	Prefix Hash Tree. 56, 59, 60
PT	Protocol Translation. 39, 42, 44
SCC	Strongly Connected Component. 43
TCA	Transitive Closure Algorithm. 31
TCM	Transitive Closure Method. 3, 24, 30
VPN	Virtual Public Network. 44
XPath	XML Path Language. 56

1 INTRODUCTION

This work presents a novel approach for distributed searches in Peer-to-Peer (P2P) overlays based on regular expressions. Data, such as peer identities, is published under a regular expression and can then be located using strings that match the regular expression. General purpose regular expressions offer a potent method for participants to advertise services or capabilities, and to enable other peers to discover partners with adequate capabilities in a fully decentralized and distributed fashion.

For example, a player might be searching for an opponent with particular availability, skills, and using hardware that offers certain minimal computational capabilities. Another example would be a scheduler that is searching for peers that are offering certain hardware and are willing to provide their computational power for distributed computing. Finally, in this thesis, we focus on the problem of finding users that are willing to provide network access using a specified protocol to a particular destination. This is useful for overlay networks that are used to tunnel IP packets to destinations that are not directly accessible for a peer in the network, for example due to protocol incompatibilities (IPv4 vs. IPv6) or firewalls.

Traditional structured Distributed Hash Table (DHT)-based P2P networks offer an efficient way of decentralized distributed storage and retrieval of data objects. In traditional DHTs, indexing is typically based on hash-keys [Bal+03]. DHTs with more expressive search capabilities exist; for example, CAN [Rat+01] can be used for range queries and P-Grid [Abe+03] provides substring, range and path queries. However, we are not aware of any existing decentralized and distributed methods for evaluating regular expressions.

Our approach implements the evaluation of regular expressions on top of traditional key-based DHTs. The usual way to evaluate a regular expression locally on one machine is to convert it to an automaton that consists of states and transitions between these states. The idea is to store each node of the automaton graph as well as the node's outgoing transitions under a certain key in the DHT. Peers that are searching for a particular string are able to evaluate the regular expressions stored in the DHT by following the transitions in the resulting Nondeterministic Finite Automaton (NFA).

This thesis presents our basic design, possible optimizations to the automaton to improve network performance and includes experimental results from an implementation in GNUnet using regular expressions to mimick AS-level routing in the P2P overlay.

The thesis is structured as follows. Chapter 2 gives an overview of the fundamental concepts and algorithms on which our design is based. The design of the overall system is described in Chapter 3, followed by Chapter 4, which focuses on the implementation. Evaluation of the implemented system is discussed in Chapter 5. Next, the work related to our approach is presented in Chapter 6. Finally, Chapter 7 gives an overview of the achievements and ideas for future work.

2 BACKGROUND

In this chapter the fundamental concepts, terminology and algorithms that are relevant to the design are discussed in detail. First the concepts of traditional key-based DHTs are introduced, followed by definitions and algorithms for finite automata and regular expressions. Furthermore optimizations for Deterministic Finite Automatons (DFAs) and the Transitive Closure Method (TCM) are presented. The chapter concludes with a short summary and a legend of symbols used throughout the thesis.

2.1 DISTRIBUTED HASH TABLES

A DHT is a data structure that allows efficient storage and look-up of data in P2P networks. It provides hash table functionality for handling (key, value) pairs. The nodes in a DHT and the stored data items are identified by a key in the same key-space. Keys are typically values from a hash function such as SHA, which provides 2^{160} (for SHA-1, SHA-2 can even provide 2^{512}) possible values, that, although possible, are unlikely to collide. Therefore nodes and items can be uniquely identified in the network. Each node maintains connections to other nodes and is responsible for storing certain (key, value) pairs, as well as routing requests to other nodes in the network. Routing efficiency is typically bound by $O(log\ n)$, meaning that for n nodes participating in the network, routing a request for the value to a key takes a maximum of $log\ n$ hops. This is achieved by imposing a structure, such as a tree, on the participating nodes. Two basic operations are provided by a DHT: $PUT(key, value)$, which stores data under a specific key in the DHT and $value = GET(key)$, which receives a previously stored value for a given key. Knowledge of the key is essential for data lookup. There is no integrated mechanism to search the stored data. In addition to routing efficiency for the PUT and GET operations, a DHT has the following characteristics [She+09].

- Scalibility, meaning with an increasing number of nodes, the overhead for managing these nodes only grows logarithmically.
- Self-organization, describing that there is no requirement for a central authority, it operates fully decentralized.
- Incremental deployability, which means a DHT works with any number of participating nodes and normal operation is still possible even when nodes constantly join and leave.

Figure 2.1 and Figure 2.2 show two schematic DHT examples. The Chord DHT structures its peers in an identifier ring, where each peer keeps a so called finger table with shortcuts to distant peers in the ring. The Kademlia DHT can be abstracted as a binary tree. It uses the XOR metric to measure the distance between two peers. There exist many other DHTs that implement a diverse set of routing algorithms and strategies, but can be abstracted using the above *PUT*/*GET* Application Programming Interface (API) abstraction.

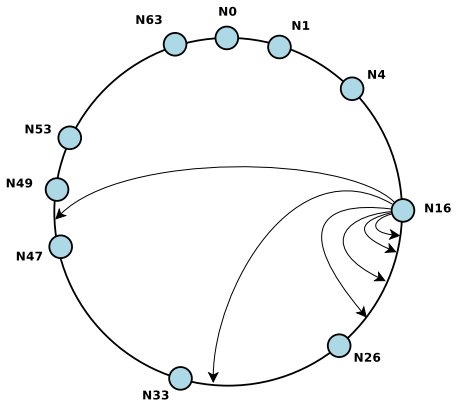


Figure 2.1: Illustration of the Chord DHT with ten nodes arranged in a circle, showing the "fingers" for node N16. Each node in Chord keeps a finger table with links to other peers, this is required to have $O(logn)$ routing performance in an n-node network. (Illustration from [She+09].)

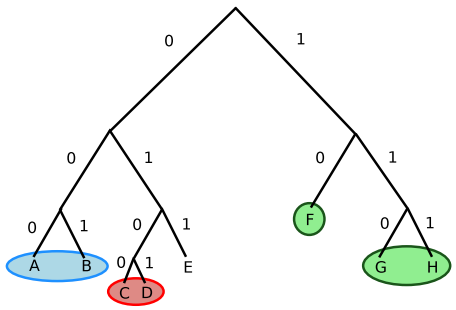


Figure 2.2: Illustration of the neighbors for peer E in the Kademlia DHT. The peer E would choose up to k peers from each of the regions as neighbors. (Illustration from [She+09].)

In this thesis we use such a traditional key-based DHT as the basis for our design. However, in addition to these concepts of traditional key-based DHTs there exist other approaches which provide more than a simple *PUT*/*GET* API. A selection of such approaches, which are relevant to this thesis, are discussed in Chapter 6.

2.2 FINITE AUTOMATA

Finite automata or Finite-State Machines (FSMs) generally consists of states, input symbols and transitions between these states. An FSM is a mathematical model of computation and can be used to model a broad spectrum of problems. [Aho+06; HMU07; Sip06]

DETERMINISTIC FINITE AUTOMATA A DFA is an FSM, basically consisting of states, input symbols and transitions between the states. A DFA is a special form of an FSM and is used to recognize languages generated by type-3 grammars from the Chomsky hierarchy (regular languages) [Cho56]. A DFA $D = (Q, \Sigma, \delta, q_0, F)$ consists of:

1. A finite set of states Q.
2. A finite set of input symbols Σ.
3. A transition function $\delta(q, a) : Q \times \Sigma \mapsto Q$ that takes a state q and an input symbol a and returns a state.
4. A start state $q_0 \in Q$.
5. A set of accepting states F, where $F \subseteq Q$.

NON-DETERMINISTIC FINITE AUTOMATA An NFA $N = (Q, \Sigma, \delta, q_0, F)$ has the same properties as the DFA, but differs in the transition function δ. For an NFA, $\delta(q, a) : Q \times \Sigma \mapsto \{q_i, ...q_k\}$ is defined as a relation that takes a state and an input symbol and returns a set of states $\{q_i, ..., q_k\} \in \Sigma$. A transition from some state $q_1 \in Q$ to $q_2 \in Q$ on some input symbol $a \in \Sigma$ is expressed as a tuple $\delta : Q \times \Sigma \times \Sigma \mapsto (a, q_1, q_2)$.

An ϵ-NFA, in contrast to an NFA, supports ϵ-transitions. This means the transition relation δ can act on the special symbol ϵ, allowing the automaton to transition from any state q_s to a state q_t without consuming any of the input symbols $\Sigma \setminus \{\epsilon\}$. Even though the definition of an NFA and an ϵ-NFA differ, the following chapters of this paper will not always strictly distinguish them. An ϵ-NFA will be generally referred to simply as an NFA.

Note that the states of a finite automaton can be freely renamed without changing the recognized language. Naming of states is only relevant for referencing certain states in the text. For example, in this document the starting state of an automaton is typically named q_0.

REPRESENTATION Both (ϵ-)NFAs and DFAs can be represented by a transition graph (see Figure 2.3 and Figure 2.4). Note that, throughout this thesis, any graphs of NFAs or DFAs do not contain all $a \in \Sigma$ at each state $q \in Q$, only the relevant edges that lead to accepting states are shown. The states are represented as graph nodes and the transition function is represented by labels on the graph's edges.

Nodes marked with a double circle are accepting states. There exists an edge with the label a from state q_s to state q_t if and only if q_t is in the return set of $\delta(q_s, a)$. In case of an ϵ-NFA the edge labels can contain ϵ transitions. According to the above definition, a DFA state cannot have two outgoing edges with the same label (because δ is defined as a function that returns exactly one state) and cannot contain any ϵ transitions.

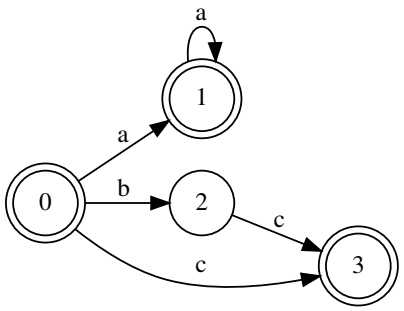


Figure 2.3: Example for a DFA graph representation, where nodes represent states, edges and their labels represent the transition function and double circled states are in the set of accepting states. This DFA accepts the empty string, the string consisting of at least one a followed by any number of optional a's and the string "bc", as well as, the string consisting of the single character c.

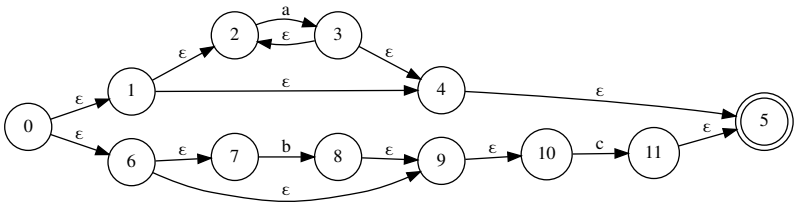


Figure 2.4: Example for an ϵ-NFA graph representation that accepts the same strings as the DFA in Figure 2.3, but includes additional ϵ-transitions that do not consume any input symbols when matching an input string.

Finite automatons can also be represented by a transition table, where table rows correspond to states and table columns correspond to input symbols. The table representation has the disadvantage of taking a great amount of space when the input alphabet Σ is large and most states do not have any transitions on most of the input symbols.

Table 2.1 shows such a table, clearly indicating the unused space taken up by ∅ symbols.

State	a	b	c	ϵ
0	∅	∅	∅	{1, 6}
1	∅	∅	∅	{2, 4}
2	{3}	∅	∅	∅
3	∅	∅	∅	{2, 4}
4	∅	∅	∅	{5}
5	∅	∅	∅	∅
6	∅	∅	∅	{7, 9}
7	∅	{8}	∅	∅
8	∅	∅	∅	{9}
9	∅	∅	∅	{10}
10	∅	∅	{11}	∅
11	∅	∅	∅	**{5}**

Table 2.1: Transition table for the δ relation, where the rows represent states $\in Q$ and the columns the input symbols $\in \Sigma$, the accepting state $\in F$ is marked bold. It indicates the unused space taken up by ∅ symbols

As stated in the beginning of this chapter, finite automata are used to recognize regular languages. If a DFA or (ϵ-)NFA accepts an input string, the input string is a word of the language it recognizes. The finite automaton accepts an inputs string s if there is a path from the start state q_0 using the transition function $\delta(q, a)$ for any symbol a in s (mind that ϵ transitions do not consume any input symbols) to an accepting state $q_f \in F$. For example consider the DFA in Figure 2.3 and the input strings "aaa", "bc" and "b", Table 2.2 shows the paths for each input string and if the string is accepted (i.e. the last state is an accepting state).

String	State Sequence	Result
"aaa"	$0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{a} 1$	accepted
"bc"	$0 \xrightarrow{b} 2 \xrightarrow{c} 3$	accepted
"b"	$0 \xrightarrow{b} 2$	rejected

Table 2.2: DFA acceptance table for the DFA shown in Figure 2.3. The first column shows the input string, the second column shows the sequence of states that has been taken in the DFA and the final column shows if the input string is either accepted or rejected

All three of the above examples (Figure 2.3, Figure 2.4 and Table 2.1) show a representation of the same automaton. They all denote finite

automatons that accept the same language. In general both NFAs and DFAs are capable of recognizing regular languages and thus can be used interchangeably [Sip06]. In order to describe a regular language with a relatively simple string, instead of constructing relatively complex automatons, regular expressions are used.

2.3 REGULAR EXPRESSIONS

NFAs and DFAs can be described using regular expressions. The advantage of regular expressions over automata is that they describe the strings which should be accepted and abstract the underlying states and transitions. All languages expressed with regular expressions are regular languages and can be used to check if a given string is part of a given language or not.

Formally, a regular expression consists of constants, strings and special operators that denote operations over these strings and constants [Sak09; HMU07]. The following sections list and describe the constants and operators used in this thesis.

2.3.1 *Constants*

Regular expression constants are the basis for building more complex expressions. For an alphabet of input symbols Σ the following constants are defined. The empty set $\emptyset$ and empty string ϵ constants are not part of the set of input symbols Σ.

Meta character	Definition
$\emptyset$	Empty set
ϵ	Empty string
$a \in \Sigma$	Set containing only the character a

Table 2.3: The constants used in regular expressions to describe a regular language.

To build more complex expressions, the constants are used together with operators defined in the next section.

2.3.2 *Operators*

For any regular expressions A and B the following basic operators are defined.

Here $^+$ and ? are simple additions to the other operators that can be expressed as seen in Table 2.4. For convenience, several more additions to the operators defined here are commonly used, but these can be reduced to the minimal required set of operators: concatenation, union and Kleene star. For example characters can be grouped by specifying

Operator	Operation	Definition
	concatenation	$AB = \{xy \mid x \in A \ \wedge \ y \in B\}$
$\mid$	union	$A \mid B = \{x \mid x \in A \cup B\}$
*	Kleene star	$A^* = \{x \mid x \in \{\epsilon, AA^*\}\}$
$^+$	repeat one or more times	$A^+ = AA^*$
?	zero or once	$A? = \epsilon \mid A$

Table 2.4: Basic set of regular expression operators used in this thesis.

ranges like $[0-5]$, which means "any of the numbers between 0 and ", which is equivalent to $(0|1|2|3|4|5)$.

OPERATOR PRECEDENCE Operators can only be applied to symbols or well-formed regular expressions. Symbols and regular expressions can be grouped using parentheses.

1. The star operator (*), the plus operator ($^+$) and the question mark operator (?) are of the highest precedence. These operators only apply to the smallest sequence of operands to their left that form a regular expression.

2. Next in precedence is the implicit concatenation operator, which groups all expressions consisting of adjacent operands that have no intervening operators.

3. The union operator is of the lowest precedence.

EXAMPLES The example automaton representations shown in Figure 2.3, Figure 2.4 and Table 2.1 can be described using the regular expression $a^*|b?c$. In English, the automatons and the regular expression describe the language that contains the empty word ϵ, all words consisting only of a's and the words bc and c.

The above example is a fairly abstract one with no real practical use, still it shows the concept of how regular expressions are used. To give a more practical example consider a program that accepts an Internet Protocol version 4 (IPv4) address as a user input. If the program wants to validate the user input, before actually using it, it can use a regular expression to test IPv4 address validity. To do that, consider the following regular expression:

```
([01]?[0-9][0-9]? | 2[0-4][0-9] | 25[0-5]).
([01]?[0-9][0-9]? | 2[0-4][0-9] | 25[0-5]).
([01]?[0-9][0-9]? | 2[0-4][0-9] | 25[0-5]).
([01]?[0-9][0-9]? | 2[0-4][0-9] | 25[0-5])
```

It will accept "**192.0.2.67**", but not the invalid "**012.345.678.9**" as an input.

Another example for practical use of regular expressions is the validation of e-mail addresses. The following regular expression[1] can be used to check if a given e-mail address is valid.

```
[a-z0-9!#$%&'*+/=?^_`{|}~-]+
(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9]
(?:[a-z0-9-]*[a-z0-9])?
```

Although this expression is already more complicated than the IPv4 example above, it is not completely compatible with the official RFC-822 standard. A fully standard compliant expression is presented in Appendix C.1.

2.4 CONVERTING REGULAR EXPRESSIONS TO AUTOMATA

Evaluating regular expressions is done by converting them to finite automata. The automata are then simulated to process an input string and decide if the given string is part of the language described by the regular expression. Our approach is to perform the evaluation of regular expressions in a distributed way that is also based on the construction of finite automata. This section describes how finite automata, in particular NFAs and DFAs, are constructed from regular expressions.

The steps to construct a DFA out of a given regular expression are:

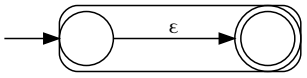
1. Parsing - parse the regular expression
2. NFA - construct an NFA from the parsed regular expression
3. DFA - convert the NFA into a DFA

2.4.1 *NFA construction*

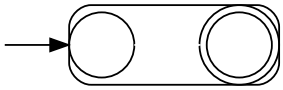
Constructing an (ϵ-)NFA for any given regular expression R is done by creating intermediate NFAs that are combined to the full NFA, accepting the same regular language as R. The basic automatons that accept the languages only consisting of either ϵ (a), $\emptyset$ (b) or one character a (c) are shown in Figure 2.5.

More complex regular expressions are then constructed using the NFA fragments shown in Figure 2.6 (R and S are itself regular expressions). These represent the operators defined in Section 2.3.2, where only the basic operations for **concatenation** (a), **union** (b) and **star** (c) are needed here, because any regular expression can be reduced to these operations.

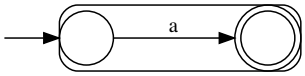
1 Regular expression for e-mail address validation taken from http://www.regular-expressions.info/email.html



(a) NFA accepting the expression ϵ



(b) NFA handling $\emptyset$



(c) NFA accepting the regular expression a

Figure 2.5: Basic constructs for building an NFA

The described NFA structures are used to construct an NFA that accepts the regular expression R, by parsing the regular expression using a stack. For each character and operation one of the basic structures is created by popping one or two elements from the stack and building a new fragment that is pushed back onto the stack. The following procedures show how the NFA fragments are created for **concatenation**, **union** and **star**.

Procedure NFA concatenation

Procedure: Takes a stack S, containing at least two NFA fragments, pops them from S, creates a new NFA fragment that concatenates the two fragments and pushes the new fragment to the stack.

Input: Stack S containing at least two NFA $n_0, n_1 \in (Q, \Sigma, \delta, q, F)$
Output: Stack S containing a new NFA fragment

1 $(Q_0, \Sigma_0, \delta_0, q_0, F_0) \leftarrow pop(S)$;
2 $(Q_1, \Sigma_1, \delta_1, q_1, F_1) \leftarrow pop(S)$;
3 $\delta' \leftarrow \delta_0 \cup \delta_1 \cup \bigcup_{q \in F_0} \{(\epsilon, q, q_1)\}$;
4 $push(S, (Q_0 \cup Q_1, \Sigma_0 \cup \Sigma_1, \delta', q_0, F_1))$;

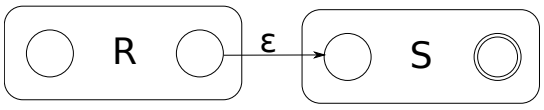
Procedure NFA union

Procedure: Takes a stack S, containing at least two NFA fragments, pops them from S, creates a new NFA fragment that joins the two fragments and pushes the new fragment to the stack.

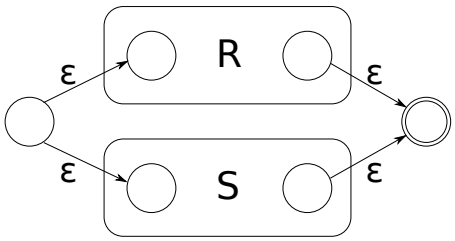
Input: Stack S containing at least two NFA $n_0, n_1 \in (Q, \Sigma, \delta, q, F)$
Output: Stack S containing a new NFA fragment

1 $(Q_0, \Sigma_0, \delta_0, q_0, F_0) \leftarrow pop(S)$;
2 $(Q_1, \Sigma_1, \delta_1, q_1, F_1) \leftarrow pop(S)$;
3 $N \leftarrow (Q_0 \cup Q_1 \cup \{q_0', q_f'\}, \Sigma_0 \cup \Sigma_1, \delta', q_0')$;
4 $\delta' \leftarrow \delta_0 \cup \delta_1 \cup \{(\epsilon, q_0', q_0), (\epsilon, q_0', q_1)\} \cup \bigcup_{q \in F_0 \cup F_1} (\epsilon, q, q_f')$;
5 $push(S, N)$;

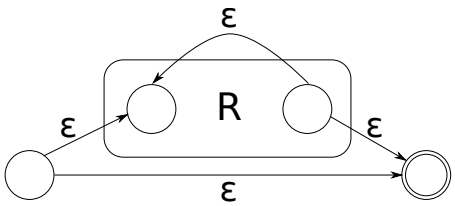
Parsing the regular expression R and building the corresponding NFA as described above can be done in parallel, so that at the end the



(a) concatenation



(b) union



(c) star

Figure 2.6: Illustration of NFA fragments for the concatenation (a) and union (b) of two regular expressions R and S and the star operation (c) for a regular expression R.

Procedure NFA star

Procedure: Takes a stack S, containing at least one NFA fragment, pops it from S, creates a new NFA star fragment and pushes the new fragment to the stack.

Input: Stack S containing NFA $n_1, ..., n_k \in (Q, \Sigma, \delta, q, F)$
Output: Stack S containing a new NFA fragment

1 $(Q_0, \Sigma_0, \delta_0, q_0, F_0) \leftarrow pop(S)$;
2 $N \leftarrow (Q_0 \cup \{q_0', q_f'\}, \Sigma_0, \delta', q_0', \{q_f'\})$;
3 $\delta' \leftarrow \delta_0 \cup \{(\epsilon, q_0', q_0), (\epsilon, q_0', q_f')\} \cup \bigcup_{q \in F_0} \{(\epsilon, q, q_f')\}$;
4 $push(S, N)$;

stack contains a single NFA that accepts the language described by R. [HMU07]

2.4.2 *DFA construction*

Constructing a DFA is done by NFA conversion [App97]. As described in Section 2.2, both NFAs and DFAs are used to recognize regular languages and can be used interchangeably. Thus it is possible to convert a NFA into a DFA. The basic idea of this conversion is that each state in the newly constructed DFA is based on a set of NFA states. Computing the sets of NFA states is done with the functions defined in the following table.

Closure function	Description
$closure(q, a)$	The set of NFA states that are reachable from state q, only by using transitions with label a. For the special case that a is ϵ, this is called ϵ-$closure(q)$ and the state q is included in this set.
$closure\_set(M, a)$	The set of NFA states that are reachable from the set $M \subseteq Q$ of NFA states using only input symbol a as the transition label.

Table 2.5: Closure functions for DFA construction

The first state of the new DFA, called q_0^{DFA}, is defined as $q_0^{DFA} = \epsilon\text{-}closure(q_0^{NFA})$, which is the set of NFA states reachable from the start state of of the NFA using ϵ-transitions only. Subsequent states are created using $\epsilon\text{-}closure(closure\_set(q_0^{DFA}, a))$ for each $a \in \Sigma$ and adding new transitions from q_o^{DFA} to the new subsequent states. If the set of NFA states that is used to create a new DFA state contains an accepting state, the new DFA state is also marked as an accepting state. This is repeated until no new DFA states are created.

EXAMPLE The above algorithm is best validated and illustrated by an example. Figure 2.7 shows an example NFA for the regular expression $a^*|b$ that is converted into it's DFA counterpart.

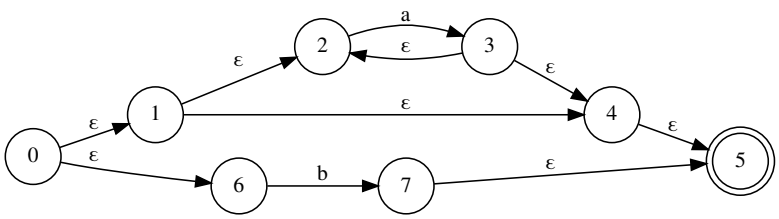


Figure 2.7: Example NFA for DFA construction that accepts the regular expression $a^*|b$.

The first step is the DFA's starting state, the ϵ-closure of the NFA's starting state: $q_0^{DFA} = \{0, 1, 2, 4, 5, 6\}$. This state is marked as an accepting state, because it includes q_5^{NFA}, which is also an accepting state. Calculating $\epsilon\text{-}closure(closure\_set(\{0, 1, 2, 4, 5, 6\}, a))$ gives the next state $q_1^{DFA} = \{2, 3, 4, 5\}$ and $q_2^{DFA} = \{5, 7\}$ is created by $\epsilon\text{-}closure(closure\_set(\{0, 1, 2, 4, 5, 6\}, b))$. Both q_1^{DFA} and q_2^{DFA} are accepting states and marked as such. From the newly created state q_1^{DFA}, the *closure_set* on a transitions to q_1^{DFA} itself. At this stage there are no more possible transitions, which means the DFA construction is

finished. The resulting automaton can be seen in Figure 2.8, showing the NFA state id's for each of the DFA states.

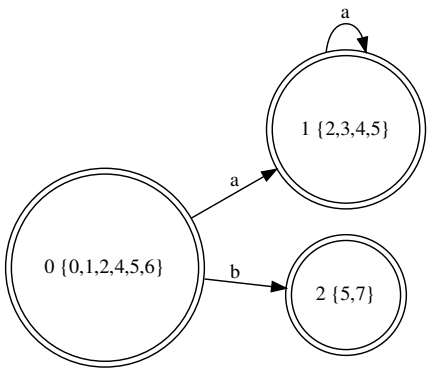


Figure 2.8: Resulting DFA that was constructed from the NFA shown in Figure 2.7. It as well accepts the regular expression $a^*|b$. Each state contains the set of NFA state ids it is based on. For example state 2 in the DFA is based on NFA states 5 and 7.

2.4.3 *DFA minimization*

For each DFA there exists an equivalent minimal DFA, which is a DFA that recognizes the same regular language, with a minimum number of states. Moreover, the minimal DFA is also unique for the language, except for the renaming of states [HMU07]. The creation of a DFA by NFA conversion itself does not guarantee that the resulting DFA is minimal. Minimization requires further steps that have to be applied to the DFA, which are described in the following paragraphs.

1. REMOVE UNREACHABLE STATES Unreachable states are states where there is no path leading from start state to the unreachable state on any of the input symbols Σ. Removing them is done by traversing the DFA from the start state and marking all visited states on each path. The states that were not marked are not reachable and therefore removed.

2. REMOVE DEAD STATES A dead state is a state that is not an accepting state and does not transition to any other state than itself. These states can be safely removed. Finding dead states is simply done by iterating over all states contained in the DFA and marking states that are not marked as accepting states, as well as do have either no transitions or only transitions to itself. All marked states are then removed.

3. MERGE NON-DISTINGUISHABLE STATES The idea of this algorithm is to find all states that can be distinguished and then merge the states which were not found to be distinguishable. This is done by first marking all pairs of states that are clearly distinguishable, which are the pairs where one state is an accepting state ($\in F$) and the other is non accepting ($\notin F$). By iteratively checking for all non marked state pairs if they can transition to a pair of states that is already marked until no new pairs are marked, all pairs of states that are distinguishable can be found. So that all pairs that have not been marked are the states that cannot be distinguished and can therefore be merged [Sino6]. This process is shown in Algorithm 1.

ALGORITHM 1: Merging non-distinguishable DFA states

ALGORITHM: This algorithm takes a DFA D as an input and first marks all clearly distinguishable pairs of states, which are the pairs, where one state is in the set of accepting states and the other states is not an accepting state. Thereafter it marks all pairs of distinguishable states and merges the states that were not marked

Input: DFA $D = (Q, \Sigma, \delta, q_0, F)$
Output: Minimized DFA D

$M \leftarrow \emptyset$;
foreach $(q', q) \in Q$ **do**
    if $q' \in F \wedge q \notin F$ **then**
        $M \leftarrow M \cup (q', q)$;
    end
end
repeat
    foreach $(q', q) \in Q \setminus M$ **do**
        foreach $a \in \Sigma$ **do**
            if $(\delta(q', a), \delta(q, a)) \in M$ **then**
                $M \leftarrow M \cup (q', q)$;
            end
        end
    end
until *M was not updated*;
foreach $(q', q) \in Q \setminus M$ **do**
    `/* Merge states q' and q. */`
    $Q \leftarrow Q \setminus \{q'\}$;
    foreach $a \in \Sigma$ **do**
        $\delta \leftarrow \delta \cup \{(q', a) \mapsto \delta(q, a)\}$
    end
    foreach $\hat{q} \in \{q_i | \forall (q_i, a) \in Q \times \Sigma \wedge q' = \delta(q_i, a)\}$ **do**
        $\delta \leftarrow \delta \cup \{(\hat{q}, a) \mapsto q\}$;
    end
end

2.5 DFA OPTIMIZATIONS

There exist several approaches to how a DFA can be optimized. Current research focuses on applications for high-throughput pattern matching in the field of Deep Packet Inspection (DPI). These approaches can be grouped in the following four basic categories.

2.5.1 *Edge compression*

Edge compression [Kum06; Bec07; BC08] exploits redundancy present in the transitions between states to minimize the number of edges in the DFA. Several transitions of the automaton are incrementally replaced with a single default transition that does not consume any input symbols. This reduces space requirements for a given DFA by eliminating redundant transitions. In [Kum06] a reduction of transitions by more than 95% is achieved. The trade-off for this space reduction is the increase of required state traversals. [Bec07] proposes an algorithm that results in at most $2n$ state visits for a matching string of length n.

2.5.2 *Increased stride*

Increasing the stride of an automaton, as proposed in [Bro06], adds new transitions to the original automaton. Algorithm 2 shows the doubling of strides for a given DFA D.

ALGORITHM 2: Double Stride algorithm by Brodie et al. [Bro06]

ALGORITHM: Doubling stride in a DFA

Input: $DFA\ D = (Q, \Sigma^k, \delta^k, q_0, F)$
Output: $DFA\ D^{2k} = (Q, \Sigma^{2k}, \delta^{2k}, q_0, F)$

```
1 foreach q ∈ Q do
2     Σ^2k ← Σ^k;
3     foreach a ∈ Σ^k do
4         foreach a' ∈ Σ^k do
5             δ^2k(q, aa') ← δ^k(δ^k(q, a), a');
6             Σ^2k ← Σ^2k ∪ {aa'};
7         end
8     end
9 end
```

Increasing the stride of a DFA has the advantage of decreasing the number of required state traversals for string matching. For stride doubling the decrease is $n/2$ for a string of length n. The disadvantage is that the space requirement for storing the additional transitions increases by $|\Sigma|^2$ in the worst case. Thus stride increasing is always a

trade-off between fewer state traversals and an exponential increase of transitions.

Figure 2.9 shows the double-strided DFA graph for the regular expression $abcde(f|g)^*$.

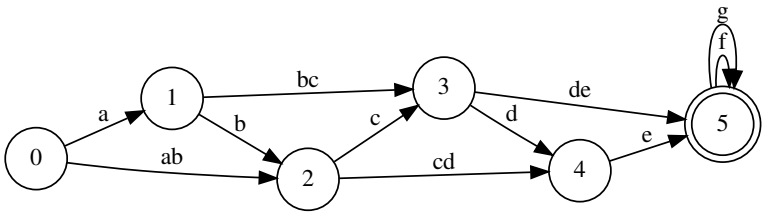


Figure 2.9: DFA graph with added double-strides for the regular expression $abcde(f|g)^*$.

2.5.3 *Path compression*

For path compression, paths with multiple transitions through multiple states are replaced with a single transition that has a string (consisting of the union of all traversed transition labels) as the label and intermediate states are removed. For example consider the following regular expression: $abc(d^*|e)fgh$. Creating a DFA for it will produce a graph with a linear structure shown in Figure 2.10.

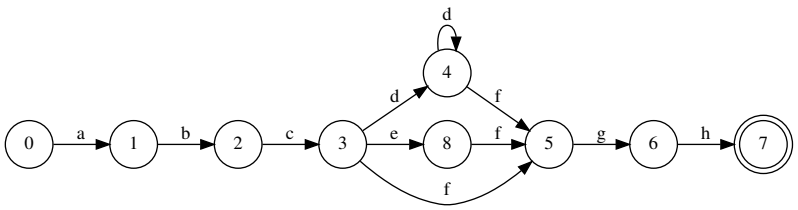


Figure 2.10: DFA graph for the regular expression $abc(d^*|e)fgh$

Applying the path compression algorithm presented in Algorithm 3 will replace the linear path consisting of multiple transitions with a single transition as illustrated in Figure 2.11.

2.5.4 *Alphabet reduction*

Alphabet reduction [Bec07; Bro06; BC08] maps sets of symbols from an input alphabet Σ to a new input alphabet Σ^* by grouping characters

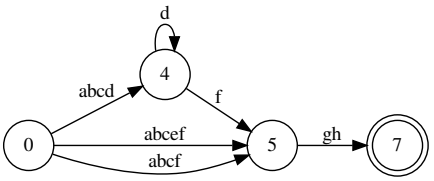


Figure 2.11: DFA graph with applied path compression for the regular expression $abc(d^*|e)fgh$

ALGORITHM 3: NFA path compression

ALGORITHM: Algorithm that compresses linear paths in the automaton

Input: NFA $N = (Q, \Sigma, \delta, q_0, F)$

Output: Path-compressed NFA N

1 $S_q \leftarrow \{q_i | q_i \in F \vee |\{q_a | (q_a, a) \in Q \times \Sigma \wedge (q_a, a) \mapsto q_i \in \delta\}| = 1\} \setminus \{q_0\}$;

2 **foreach** $q \in S_q$ **do**

3 $\quad \delta_{new} \leftarrow \bigcup_{(q_k,l)\mapsto q \in \delta \wedge (q,m)\mapsto q' \in \delta} \{(q_k, l+m) \mapsto q'\}$;

4 $\quad \delta_{dead} \leftarrow \bigcup_{(q_k,l)\mapsto q \in \delta \wedge (q,m)\mapsto q' \in \delta} \{(q_k, l) \mapsto q, (q, m) \mapsto q'\}$;

5 $\quad \delta \leftarrow \{\delta \cup \delta_{new}\} \setminus \delta_{dead}$;

6 **end**

7 $Q \leftarrow Q \setminus S_q$;

8 $\Sigma \leftarrow \{a | (q, a) \mapsto q' \in \delta\}$;

that label the same transitions in the automaton. In the worst case $\Sigma^* = \mathcal{P}(\Sigma)$, but depending on the actual implementation $|\Sigma|$ can be reduced, so that $|\Sigma^*| < |\Sigma|$. The idea is to partition Σ into classes of symbols $C_1, \ldots, C_k$, where each class contains the input symbols that are indistinguishable in the sense of DFA operations. Especially, given a DFA $D = (Q, \Sigma, \delta, q_0, F)$ then two input symbols $a, b \in \Sigma$ are in the same class C_i if and only if $\forall q_i \in Q : \delta(q_i, a) = \delta(q_i, b)$

For example, consider a state that has three outgoing transitions with three different labels to another state. These three transitions can be replaced by one transition that has a new label in Σ^*, which is not in the original input alphabet Σ. This is then repeated for all other states that have the same three outgoing transitions to another state. This process is illustrated in Figure 2.12.

In order to use the new DFA for string matching, a translation table that maps labels from Σ^* to sets of labels from Σ has to be provided. Table 2.6 shows the translation table for the above example.

(a) DFA that recognizes the regular expression $(a|b|c)x(y|a|b|c)$ and has an input alphabet $\Sigma = a, b, c, x, y$ where $|\Sigma| = 5$.

(b) Alphabet reduced DFA with a new input alphabet $\Sigma^* = x, y, z$ and $|\Sigma^*| = 3$, where the transitions for a, b, c have been replaced with a new transition z

Figure 2.12: Example for alphabet reduction on a DFA where the size of the input alphabet is reduced from 5 to 3 and the number of transitions in the graph have been reduced by 50%.

Σ^*	Σ
z	$\{a, b, c\}$
v	$\{y, z\}$

Table 2.6: Alphabet reduction transition table that is needed for string matching with the alphabet-reduced DFA in Figure 2.12(a).

2.6 TRANSITIVE CLOSURE METHOD

The transitive closure algorithm [HMU07; Neu05] is a method for calculating regular expressions from a DFA or NFA. We describe this method here as it will be used as a key building block in our main algorithm. The transitive closure algorithm incrementally constructs a regular expression for each pair of states in a DFA. For example, consider the simple DFA D shown in Figure 2.13.

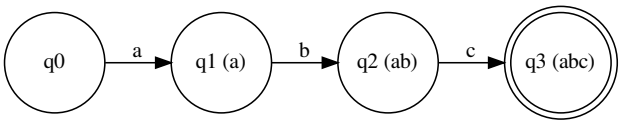


Figure 2.13: Simple DFA for the regular expression abc, showing the (partial) regular expressions (a, ab, abc) for each state that are used as state identifiers.

The regular expression for the transition from q_0 to q_1 is simply a, for q_1 to q_2 the expression is b. In addition the regular expression from q_0 to q_2 is the concatenation of the above sub-expressions and therefore becomes ab. The complete regular expression for D is then abc. In order to get the regular expression for each state in the automaton, the concatenation of all sub-expressions on the paths leading to the respective state has to be build. More generally suppose that for some DFA A with n states the states are labeled $\{q_1, q_2, \ldots, q_n\}$. Then $R_{ij}^{(k)}$ can be defined as the set of regular expressions on the path

between states q_i and $q_j \in A$, with the restriction that this path has no intermediate state $q_m \in A$ with $m > k$. Starting with $k = 0$, i.e. all paths in the automaton that do not have any intermediate states, there are only two possibilities:

1. A transition from q_i to q_j with some label w.
2. A self-transition consisting only of some state q_i.

If $i \neq j$ only the first case is possible and $R_{ij}^{(0)}$ is defined as follows:

$$R_{ij}^{(0)} = \begin{cases} \emptyset & \text{if there is no transition from } q_i \text{ to } q_j \\ a & \text{if } a \text{ transitions from } q_i \text{ to } q_j \\ a_1|a_2|\ldots|a_k & \text{for all } k \text{ transitions from } q_i \text{ to } q_j \end{cases} \tag{2.1}$$

On the other hand if $i = j$ than the path of length zero and all loops from q_i to itself are allowed. This defines $R_{ij}^{(0)}$ as follows:

$$R_{ij}^{(0)} = \begin{cases} \epsilon & \text{if there is no transition from } q_i \text{ to } q_j \\ \epsilon|a & \text{if } a \text{ transitions from } q_i \text{ to } q_j \\ \epsilon|a_1|a_2|\ldots|a_k & \text{for all } k \text{ transitions from } q_i \text{ to } q_j \end{cases} \tag{2.2}$$

For $R_{ij}^{(k)}$ where $0 < k <= n$ there are again two possible cases to look at.

1. The path from q_i to q_j does not traverse through any state q_m where $m > k$ and therefore the expression for this path is recursively defined as $R_{ij}^{(k-1)}$.
2. The path from q_i to q_j traverses through state q_k at least once. In this case the path can be divided into three parts as shown in Figure 2.14. The first part transitions the automaton from q_i to q_k without passing through q_k, the last part transitions from q_k to q_j without passing through q_k. The middle part is the path from q_k to itself, zero, once or more times.

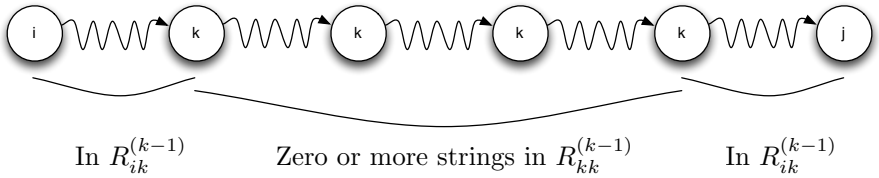


Figure 2.14: Illustration of the path between states i and j through state k (Illustration from [HMU07]).

Combining the above two cases results in Equation (2.3). Which is the recursive definition for all $R_{ij}^{(k)}$ with $0 < k <= n$. Constructing

$R_{ij}^{(n)}$ for all $q_i, q_j \in A$ leads to all (partial) regular expressions that are needed to identify the sates of the DFA A.

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} \mid R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)} \tag{2.3}$$

The overall regular expression for A is the union of all $R_{0j}^{(n)}$ ($i = 0$ is the starting state) where $j \in F$ (i.e. j is an accepting state). Moreover the partial regular expression for each state q_j simply is $R_{0j}^{(n)}$ (i.e. the regular expression for the path from the starting state to q_j). The complete algorithm is shown in Algorithm 4. It takes a DFA A with states numbered $0, \ldots, n-1$ and $n = |Q|$ as the input and assigns a regular expression to each state in A. Because $R^{(k)}$ only depends on $R^{(k-1)}$, all other expressions can be discarded and therefore $R\_last = R^{(k-1)}$ and $R\_cur = R^{(k)}$. Algorithm 4 uses an additional function, named `simplify`, which is discussed in Section 2.6.1.

ALGORITHM 4: Transitive closure algorithm

ALGORITHM: Computes a (partial) regular expression for each state in the given DFA.

Input: DFA $D = (Q, \Sigma, \delta, q_0, F)$ with states $q_j \in Q$ for $j \in \{0, \ldots, n-1\}$

Output: Regular expressions $R_{ij}^{(n)}$ for $i, j \in \{0, \ldots, n-1\}$

```
1  foreach i ∈ {0,...,n−1} do
2      foreach j ∈ {0,...,n−1} do
3          if i = j then
4              R_ij^(0) ← ϵ;
5          else
6              R_ij^(0) ← ∅;
7          end
8          foreach a ∈ Σ ∧ q_j = δ(q_i, a) do
9              R_ij^(0) ← R_ij^(0) | a;
10         end
11     end
12 end
13 foreach k ∈ {0,...,n−1} do
14     foreach i ∈ {0,...,n−1} do
15         foreach j ∈ {0,...,n−1} do
16             R_ij^(k+1) ← simplify(R_ij^(k) | R_ik^(k) (R_kk^(k))* R_kj^(k));
17         end
18     end
19 end
```

The algorithm for computing regular expressions for each state in a given automaton is expensive with a complexity of $O(n^3)$. Looping over all $n = |Q|$ states of the automaton for $k, i, j \in \{0, \ldots, n-1\}$. [HMU07] describes another, less expensive algorithm that is based on removing states of the automaton and focuses on the construction of a regular expression for the whole DFA. Therefore it would have

to be repeated n times for each state in the automaton to generate an expression for each of the states, which is a requirement for the approach of this work.

Another aspect of the transitive closure method are the space requirements. On average the length of an expression can grow by a factor of 4 (see Equation (2.3)). In the worst case, constructing the basis for the iteration, $R^{(0)}$, will lead to the union of all input symbols $|\Sigma|$. With $n = |Q|$ iterations the expressions can expand on the order of 4^n, resulting in a worst case space complexity of $O(|\Sigma| 4^n)$. This will cause problems in the implementation, where memory is limited. Thus a simplification of the regular expression at each step in the above algorithm is important.

2.6.1 *Regex simplification*

Building regular expressions from DFAs using the transitive closure method described above is based mainly based on Equation (2.3). Without any simplification of this equation, the space requirements of the final regular expression, as well as the expressions used for state identification, will quickly exceed the available memory. Moreover, simplifying the final regular expressions is not enough. Therefore simplification has to take place directly at each step in the above algorithm by rewriting Equation (2.3). This is done by using the knowledge of the basic structure and applying general algebraic rules for regular expressions at each step of the iteration. The following definitions show the main algebraic laws, where R, S and T are each some regular expression.

- **Identity**
  Union: $(\emptyset \mid R) = (R \mid \emptyset) = R$
  Concatenation:
  $(\epsilon R) = (R\epsilon) = R$
  $(\emptyset R) = (R\emptyset) = \emptyset$

- **Associativity**
  Union: $R \mid (S \mid T) = (R \mid S) \mid T$
  Concatenation: $R(ST) = (RS)T$

- **Commutativity**
  $R|S = S|R$
  Note that concatenation is not commutative.

- **Distributivity**
  $R(S \mid T) = (RS) \mid (RT)$
  $(S \mid T)R = (SR) \mid (TR)$

- **Idempotency**
  $R \mid R = R$

Where $\boldsymbol{\epsilon}$ is the **identity** for **concatenation** and $\emptyset$ is the **identity** for **union**, as well as, the **annihilator** for **concatenation**.

To better illustrate application of algebraic rules to Equation (2.3), it is rewritten to the following simpler form.

$$\underbrace{R}_{R_{ij}^{(k)}} = \overbrace{\underbrace{L}^{R_{left}} \mid \overbrace{M\ (N)^*\ P}^{R_{right}}}^{R_{ij}^{(k-1)}\ \mid\ R_{ik}^{(k-1)}\ (R_{kk}^{(k-1)})^*\ R_{kj}^{(k-1)}} \tag{2.4}$$

These definitions already allow for some optimization of Equation (2.4). Applying the rule for idempotency, if R_{left} equals R_{right}, R can be simplified to L. Furthermore applying the rule for identity of union, if either R_{left} or R_{right} is $\emptyset$ it can be simply omitted.

In addition to the above rules, [HMU07] defines the following algebraic rules involving closures.

- $(S^*)^* = S^*$
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $S^+ = SS^* = S^*S$
- $S^* = S^+|\epsilon$

The defined general algebraic laws for regular expressions build the foundation for defining specific rewriting rules that are used to replace complex regular expressions of the form $R = L \mid M\ (N)^*\ P$ to a simpler representation. Table 2.7 shows an excerpt with some important rewrite rules used.

The goal is to simplify regular expressions at each step of the transitive closure algorithm. This implies that rewriting regular expressions of the form $L = M\ (N)^*\ P$ has to be efficient, because this is executed at the inner loop of an already expensive algorithm, and still fulfill the space requirements. Using the described method of defining and applying algebraic rules that replace expressions is of complexity $O(1)$ while still significantly shortening the complex expressions. This, of course, heavily depends on the input DFA.

Result	Rewrite rule for $R = L \mid M\ (N)^*\ P$
S^*	$S \mid (\epsilon \mid S)\ S^*\ (\epsilon \mid S)$
	$S \mid (\epsilon \mid S)\ (\epsilon \mid S)^*\ (\epsilon \mid S)$
	$(\epsilon \mid S) \mid S\ S^*\ S$
	$(\epsilon \mid S) \mid S\ S^*\ (\epsilon \mid S)$
	$(\epsilon \mid S) \mid (\epsilon \mid S)\ S^*\ S$
	$(\epsilon \mid S) \mid (\epsilon \mid S)\ S^*\ (\epsilon \mid S)$
	$(\epsilon \mid S) \mid (\epsilon \mid S)\ (\epsilon \mid S)^*\ (\epsilon \mid S)$
S^+	$S \mid S\ S^*\ S$
	$S \mid (\epsilon \mid S)\ S^*\ S$
	$S \mid S\ S^*\ (\epsilon \mid S)$
	$S \mid (\epsilon \mid S)\ (\epsilon \mid S)^*\ S$
	$S \mid S\ (\epsilon \mid S)^*\ (\epsilon \mid S)$
ST^*	$S \mid ST^*T$
T^*S	$S \mid TT^*S$

Table 2.7: Excerpt from regular expression rewriting rules

2.7 SUMMARY

This chapter gave an overview of the relevant fundamental concepts, terminology and algorithms. First the concepts of DHTs, which provide a distributed key-value storage, was introduced. Moreover the terminology of finite automata was presented and how they can be constructed from regular expressions. Central DFA optimizations where presented, which essentially always imply a certain trade-off between gains and losses of performance and memory consumption. Finally the TCM was presented, which is a key foundation for the central algorithm of this thesis.

The following table gives an overview and provides a central reference of the important symbols used throughout this thesis. It includes descriptions of symbols that will be introduced later, in the following chapters.

Symbol	Description
D, N	Finite automata, DFA D and NFA N.
Q	A set of states, where states are usually referred to as q
q_0	Start state of a DFA or NFA, also referred to as q'_0
Σ	Input alphabet alphabet of a DFA or NFA
δ	Transition function of a DFA or transition relation of an NFA
F	Set of final states of a DFA or NFA
a	Input symbol $a \in \Sigma$
ϵ	Empty word
$\|$, *, $^+$, ?	Regular expression operators, defined in Table 2.4
R, S, T	Regular expressions
S	Stack, for which the operations *push*, *pop*, *empty* are defined to add an element on top of the stack, retrieve and remove and element from the top of the stack and check if the stack is empty.
R_{ij}	Set of regular expressions between all n states of an NFA or DFA where states are indexed $0 <= i, j < n$
m	Merge bijection between sets of states (see Definition 1)
k	Length of the initial key for decentralizing the start state

Table 2.8: Reference table for symbols used throughout the thesis

3 DESIGN

This chapter lays out the design of our approach to support decentralized evaluation of regular expressions on DHT-based overlays.

Offerers of services describe them using regular expressions. First, each regular expression is converted to a DFA by the offerer. The DFA is then published in the DHT. Specifically, each of the automaton's states, as well as its outgoing transitions are stored under a certain key in the DHT. Additionally, the identity of the participant is stored under the keys that correspond to accepting states in the automaton. As multiple offerers store DFAs in the DHT, the resulting structure in the DHT is an NFA.

Patrons search for a particular service using a string. They follow the transitions of the resulting NFA to learn the identities of offerers with matching services. Patrons begin their search at a starting state by retrieving the corresponding data from the DHT. They then match the edges of the state against their search string, obtaining a fresh set of states (and corresponding keys) to process until the full search string is matched and an accepting state is reached. As the DHT contains an NFA, the number of DHT GET operations that needs to be performed is not bounded by the length of the search string.

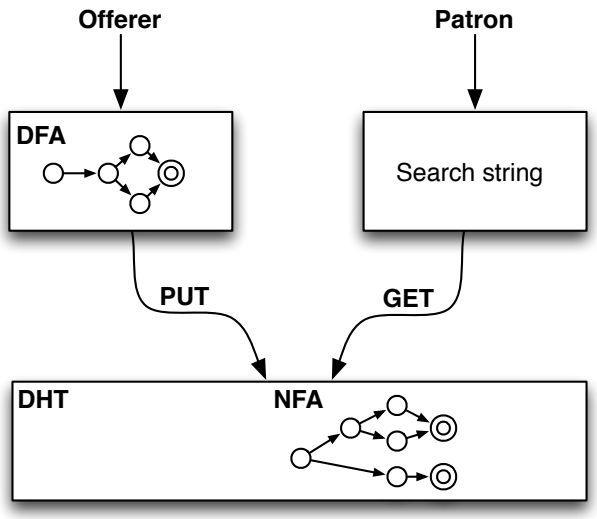


Figure 3.1: Schematic overview of the presented approach. An offerer that is offering certain capabilities compiles a regular expression into a DFA and stores it in the DHT. A patron searching for a particular string traverses the NFA stored in the DHT to find matching offerers.

3.1 MAPPING OF STATES TO KEYS

The description above does not explain how exactly DFA states are mapped to keys for the DHT. This is a central problem as the mapping has a significant impact on network performance. For example, suppose the initial state is mapped to a well-known key and each offerer maps each state of his DFA to a (fresh) random key. Then the initial state in the DHT would have n (largely non-deterministic) transitions for n offerers. This would both be terrible for DHT performance for this key and require patrons to perform a very large number of DHT lookups as they traverse the NFA that begins with massive branching. Thus, a mapping of DFA states to DHT keys is required that minimizes the creation of additional edges from the same source with the same label to a different target state in the DHT.

3.1.1 *Illustrative Examples*

We will now illustrate some of the key considerations using concrete examples.

The simplest problem is that of having two regular expressions, for example $aa^*|b$ and $b|a^+$, that describe the same language and are thus equivalent. As these two expressions are equivalent, they produce equivalent automata and the different corresponding states in the automata should be mapped to the same keys.

Another problem is the desired merging of states from two or more regular expressions that begin with equivalent prefixes. For example, consider the two simple regular expressions ab and ac and the corresponding DFAs in Figure 3.2.

(a) DFA accepting ab (b) DFA accepting ac

Figure 3.2: DFAs for the regular expressions ab (a) and ac (b) that should be merged in the DHT.

When inserting these two DFAs into the DHT there are two possible outcomes, shown in Figure 3.3.

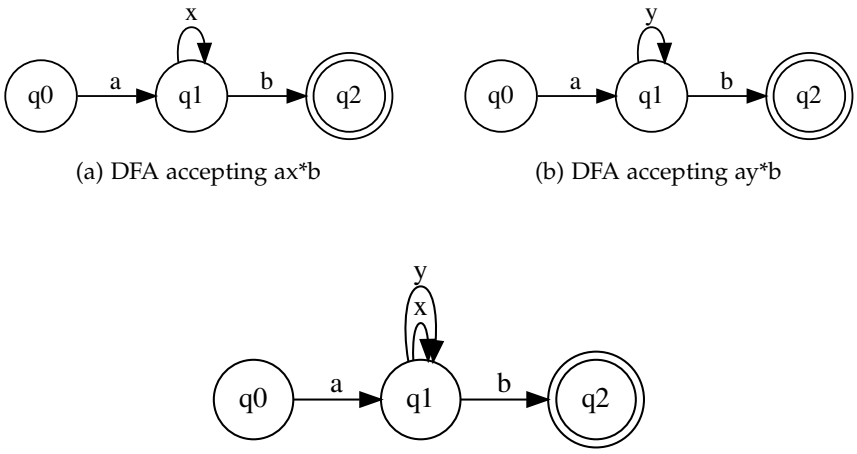
Both of the resulting automata accept the same language, but Figure 3.3 (a) is an NFA and Figure 3.3 (b) is a DFA. Producing the DFA is desirable as the DHT will have fewer transitions to store and patrons will need significantly fewer costly network operations to evaluate the DFA.

(a) Merged NFA accepting *ab|ac*

(b) Merged DFA accepting *ab|ac*

Figure 3.3: Two possibilities for merging the DFAs for the regular expressions *ab* and *ac*.

However, not all structurally similar regular expressions can be safely merged. Consider the regular expressions ax^*b and ay^*b and the corresponding DFAs in Figure 3.4.



(a) DFA accepting ax*b

(b) DFA accepting ay*b

(c) Incorrectly combined DFA for the expressions ax*b and ay*b, accepting a(x|y)*b

Figure 3.4: Example for an incorrect merge of two DFAs that does not result in the union of the two languages of the individual DFAs and thus must be avoided.

The merge illustrated in Figure 3.4 is incorrect, as the resulting DFA will accept strings (such as *axyxyb*) which are not legal for either of the regular expressions. Hence, merging of states that would result in such incorrect matches has to be prevented. Figure 3.5 shows a correctly merged automaton for ax^*b and ay^*b. Here, the merging of the two DFAs creates an NFA.

Note that a DFA for $ax^*b|ay^*b$ *exists*. However, as the two offerers cannot know about each other, it is not practical to construct such a DFA in a distributed system. In particular, as both offerers may concurrently publicise their regular expressions, attempts to lookup

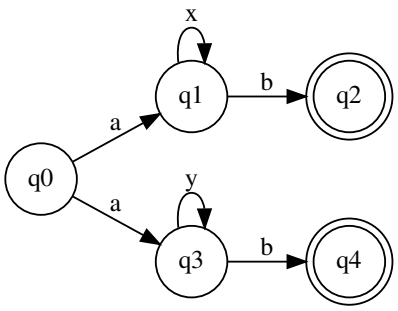


Figure 3.5: Correctly combined DFAs with separation of SCCs, resulting in an NFA that accepts the union of the languages of the two individual DFAs.

existing DFAs in the DHT cannot be guaranteed to succeed. Additionally, updating records in the DHT is generally undesirable due to the need for locking, the DHT performing distributed caching and the possibility of security implications[1]. In contrast, supplementing records by adding additional edges is much less problematic for the DHT.

3.2 PROBLEM STATEMENT

Before presenting the algorithm to merge DFAs to create a distributed NFA, we will first formally define the requirements for the operation. Our definition of a *feasible merge* of a DFA with an NFA ensures that the resulting NFA accepts the correct language, and that the operation can be executed in a distributed computer system without consistency.

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

$$\hat{\delta}(q, a) := \delta(q, a) \cup \{m(\delta'(m^{-1}(q), a))\} \tag{3.1}$$

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

$$\hat{F} := F \cup m(F' \cap M') \cup (F' \setminus M'). \tag{3.2}$$

Note that the above definition of a feasible merge operation using the bijection m excludes the possibility of creating new states — existing states and edges in the NFA are always preserved.

The CAP theorem [GL02] states that a distributed computer system cannot satisfy consistency, availability and partition tolerance at the same time. As availabiltiy and partition tolerance are hard requirements for our application domain, our defintion of an *independent*

1 An update mechanism might enable evil peers to delete information.

feasible merge requires that the feasible merge operation does not require consistency. It must be possible for many merge operations to be executed in parallel (the resulting NFA is still well-defined, as the $\cup$ operation is associative and commutative). We define an independent feasible merge as follows:

Definition 2 (Independent merge operation)**.** *An independent merge operation is a merge operation that computes a feasible merge in a distributed system and permits the concurrent execution of other independent merge operations (without knowledge or communication between the independent 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 characterization of an independent merge operation creates the necessary and sufficient conditions to be placed on a decentralized algorithm for mapping DFA states to DHT keys.

3.3 MAPPING DFA STATES TO DHT KEYS

We will now present an algorithm to compute the maximal feasible merge bijection (Algorithm 5), that is the largest (in number of mapped elements) bijection that represents a feasible merge. The basic idea is to first derive the regular expression R_{0j} between the start state q_0 and each other state $q_j \in Q$ and then add a mapping for states with equal regular expressions to the bijection. Deriving regular expressions is done using the TCM (Algorithm 4) described in Section 2.6.

In order to make regular expressions easy to compare for equivalence, we need to preprocess the DFA to ensure deterministic processing by the algorithm. Specifically, the transitive closure algorithm relies on a numbering of the DFA states. The order of numbering is not important when simply computing a regular expression that accepts the same language as the given DFA, but when using these (partial) 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.

Figure 3.6 illustrates an example DFA that has (partial) regular expressions assigned to each state.

The numbering of states in this example is important. For example, if renaming state q_1 to q_2 and q_2 to q_1 the expression for q_3 will be computed as $(cd|ab)e^*$, which is not a wrong partial regular expression for that state, but will lead to a different string. Solving this problem is done 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.

Starting with a canonical DFA, we can now describe the algorithm for executing a maximal feasible merge that satisfies Definition 1.

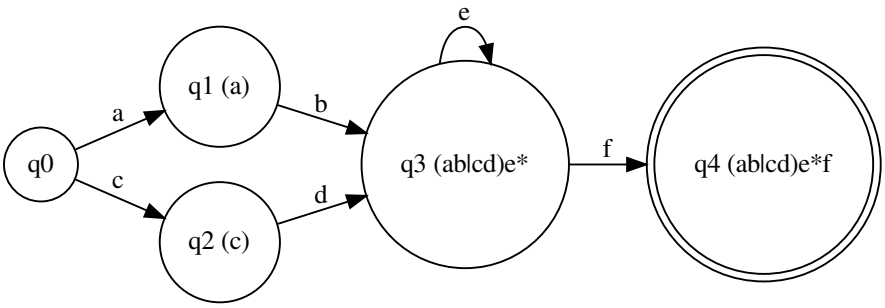


Figure 3.6: DFA graph for the regular expression "(ab | cd)e*f" with partial regular expressions assigned to each state.

ALGORITHM 5: 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. It uses the Transitive Closure Algorithm (TCA) presented in Algorithm 4

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 $R_{ij} \leftarrow \texttt{TCA}(N)$;
2 $R'_{ij} \leftarrow \texttt{TCA}(D)$;
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**

Lemma 1 (Feasible merge). *Algorithm 5 computes a feasible merge.*

Proof. First, we show 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. Hence $R_{00} = R'_{00'}$ and thus the algorithm would include $m(q'_0) \mapsto q_0$ in the bijection.

Next, we need to show that the merged NFA $(Q \cup (Q' \setminus M'), \Sigma \cup \Sigma', \hat{\delta}, q_0, \hat{F})$ accepts exactly the union of the two original languages. Suppose a word A is accepted by the NFA via some sequence of transitions $\tau \in (Q \times \Sigma)^n$ with $q_i \in \delta(q_{i-1}, a_i)$ for $(q_i, a_i) \in \tau$ and $q_n \in F$. Then the union in Equation (3.1) ensures that those transitions still exist in $\hat{\delta}$. Suppose a word B is accepted at state $q'_{n'} \in F'$ by the DFA via some sequence of transitions $\tau \in (Q' \times \Sigma')^{n'}$ with $q'_i \in \delta'(q'_{i-1}, a'_i)$ for $(q'_i, a'_i) \in \tau'$. Then for $i \in [1, \ldots, n']$ the sequence $\hat{\tau}_i := (m(q'_i), a'_i)$ is an equivalent sequence of transitions $\hat{\tau} \in (\hat{\Sigma} \times \hat{Q})^{n'}$ with $m(q'_i) \in \hat{\delta}(m(q'_{i-1}), a'_i)$ for $(m(q'_i), a'_i) \in \hat{\tau}$ and $m(q'_{n'}) \in \hat{F}$.

Finally, we need to show that the merged NFA does not accept any word that is not accepted by the original NFA or the DFA. Suppose there exists a word A which is accepted by the merged NFA but

not the original NFA or DFA. Then this word must correspond to a sequence of transitions $\hat{\tau} \in (\hat{\Sigma} \times \hat{Q})^n$ such that $\hat{q_i} \in \hat{\delta}(\hat{q_{i-1}}, a_i)$ for $(\hat{q_i}, a_i) \in \hat{\tau}$. As A was not accepted by the original NFA N, it must either end at an accept state $q_n \in \hat{F} \setminus F$ or there must exist a smallest index j for which the transition $\hat{\tau_j}$ does not exist in the original NFA N.

1. If $q_n \in \hat{F} \setminus F$, then the DFA D must accept A at state $m^{-1}(q_n) \in F'$. (as q_n and $m^{-1}(q_n)$ correspond to a pair of equivalent regular expressions R_{0n} and R'_{0n}, they must accept the same language in the DFA D and the merged NFA; note further that the construction preserves the edges and thus the subgraph isomorphism induced by m).

2. Suppose there exists a smallest index j for which the transition $\hat{\tau_j}$ does not exist in N. Then $\hat{q_j} \notin \delta(q_{j-1}, a_j)$ must hold. Given the construction of $\hat{\delta}$ (by Equation 3.1), this implies that $\hat{q_j} \in m(\delta'(m^{-1}(q_{j-1}), a_j))$. As the algorithm merges states with equivalent regular expressions, $m^{-1}(q_j) \notin M'$ (if an equivalent regular expression existed in the NFA, then j would not have been the smallest index j for which $\hat{\tau_j}$ does not exist in the original NFA). We now show by induction that A has to be accepted by the DFA. If $m^{-1}(q_j)$ is an accept state, A is clearly accepted by the DFA. If not, $\hat{q_j} \notin \hat{F}$ as $q_j \notin Q$ implies that $q_j \notin F$. Then, the fact that A was accepted by the merged NFA means that there is an transition $\hat{\tau}_{j+1}$, which again from the construction of $\hat{\delta}$ must have come from the DFA, implying a transition $q'_{j+1} \in q'(m^{-1}(q_j), a_{j+1})$. As A is a finite word, the induction must end with a corresponding accept state in F'. Thus A is accepted by the DFA D.

A must thus have been accepted by the DFA D, contradicting our assumption that A was a word not accepted by the DFA. Thus, no word exists that is accepted by the merged NFA, but not accepted by either the DFA D or the original NFA N. □

Lemma 2 (Independent merge). *Algorithm 5 directly leads to an independent merge operation to compute a feasible merge.*

Proof. The algorithm states how to derive a bijection $m \in M' \to M$. Combined with Equation (3.1), the algorithm is easily turned into a decentralized independent merge operation as follows.

A independent merge operation first computes canonical regular expressions R'_{0j} for each state $q'_j \in Q'$. Then, the operation hashes 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 in any order.

The complete merge operation is clearly 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. It is based on Algorithm 5 and thus computes a feasible merge. □

Theorem 1 (Maximal feasible merge)**.** *Algorithm 5 computes the maximal feasible merge that can be executed using an independent merge operation.*

Proof. With the previous lemmas, we only need to show that the merged NFA is maximal, that is, adding an additional element $\overline{q'}$ to M' would not result in a feasible merge. We will give a proof by contradiction. Suppose there exists an element $\overline{q'_k} \notin M'$ and a value $\overline{q_k} \notin M$ such that $\overline{m} := m \cup \{\overline{q'_k} \mapsto \overline{q_k}\}$ is still a feasible bijection.

Furthermore, note that $R_{0k} \neq R'_{0k'}$ as otherwise Algorithm 5 would have already added $q_{k'} \mapsto q_k$ to the bijection m. As the regular expressions have been canonicalized, $R_{0k} \neq R'_{0k'}$ means that the respective languages are different. We distinguish two cases (both of which can be true, and one of which must be true):

1. The languages are different because there exists a word $A \in R_{0k} \setminus R'_{0k'}$. As neither the canonicalized DFAs nor the resulting NFA contain any states that are not between the start state and an accept state, there must exist a word A from q'_0 to $\overline{q'}$. Note that we required a feasible merge algorithm to not make any assumptions about the existing NFA N (due to the lack of consistency). Thus, we can assume that there exists a word B' corresponding to a path from $\overline{q}$ to some accept state $\overline{q}_b \in \overline{F}$ such that AB' is not accepted by D or N. However, the word AB' will be accepted by the merged NFA. Thus $\overline{m}$ is not (always) a feasible merge.

2. The languages are different because there exists a word $A' \in R'_{0k'} \setminus R_{0k}$. Note again that A' can be chosen freely as details about the NFA are not available to an independent merge operation. Furthermore, suppose B is the word corresponding to the path from $\overline{q'}$ to some accept state $\overline{q'}_{b'} \in F'$. As an algorithm executing an independent merge operation cannot know all possible words A', it cannot guarantee that $A'B$ exists in D or N. However, the word $A'B$ will be accepted by the merged NFA. Thus $\overline{m}$ is not (always) a feasible merge.

In conclusion, the merged NFA is maximal as any independent merge operation with its restricted knowledge of the existing NFA cannot ensure that adding additional elements to the bijection would still always result in a feasible merge. □

3.4 DECENTRALIZING THE START STATE

When mapping the DFA to DHT keys as described so far, all starting states of all DFAs would correspond to the same regular expression — the empty string — and thus be mapped to the same key in the DHT. As a result, the nodes responsible for this particular key would be expected to experience significant load, both in terms of GET and PUT operations and the result set might also be rather large ($O(|\Sigma|)$).

We solve this problem by using the first k characters of any string in the language corresponding to the regular expression to determine an initial set of keys. Specifically, if a regular expression matches several different prefixes for the first k characters, the peer has to insert a start state under the hash of each prefix into the DHT. Those synthetic start states are then connected to the rest of the DFA as necessary.

The searching peer can now hash the first k characters of the string that should be matched, and look up the information stored under this hash to find the entry point of the corresponding NFA.

3.4.1 *Example*

For example, consider the regular expression abc^*defg^*h and $k = 4$. The entry states for this example are *abcc*, *abcd* and *abde*. Figure 3.7 shows the graph that is then inserted into the DHT. The original starting state q_0, which corresponds to the empty string, will not be inserted. The synthetic start states are highlighted using a dashed line.

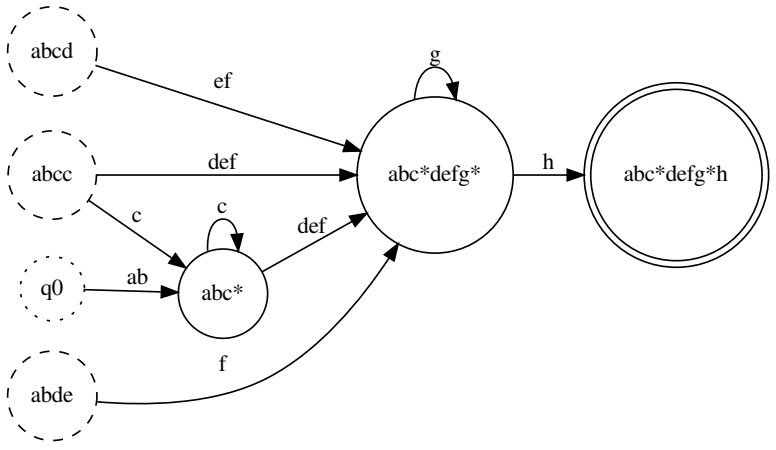


Figure 3.7: DFA graph illustrating the concept of synthetic start states, which are highlighted with a dashed line. The original start state q_0, which corresponds to the empty string, will not be inserted into the DHT.

This strategy can lead to an exponential increase in the number of synthetic entry states. For an input alphabet Σ and k initial characters, this leads in the worst case to $|\Sigma|^k$ additional states. This initial states explosion is effectively prohibited by the design of our policies.

3.4.2 *Policies*

Our implementation standardizes a particular prefix format that should be used by all applications using the service discovery infrastructure. The prefix begins with a unique application identifier followed by a version number to indicate compatibility between different revisions of the same application. This ensures that DFAs from different applications are kept separate. The prefix is followed by a certain number of bytes of padding. The actual application-specific regular expression is specified after the padding. Within an application and version combination, semantics of the strings described by the regular expression are supposed to be fixed to ensure compatibility. We refer to the complete string with application identifier, version, padding and the actual regular expression as a *policy*.

Table 3.1 gives some examples for policies. The first policy is for the "GNUNET-VPN" application in version "0001". The regular expression then describes a service for IPv4 and TCP and the 192.0.2.*-subnet. Here, "GNUNET-VPN" in version 1 specifies that a service is described by first specifying the IP version, then the transport protocol and finally the IP prefix for which the service is offered.

App. ID	Version	Padding	Regular Expression
`GNUNET-VPN`	`0001`	`0000000`	`V4-TCP-192.0.2.([0-9])*`
`COMP-POWER-NET`	`0001`	`000`	`GAME1-GHZ(0|1|2|3)`

Table 3.1: Example policy description, consisting of a fixed-length prefix that includes an application identifier, a version string and some padding, followed by the actual regular expression.

The padding in a policy is used to *control* the possible exponential growth in initial states. Given a fixed value of k that is a bit larger than the number of tokens used for the application identifier and the version number, each application can choose a padding value that results in (roughly) the *desired* number of initial states. For example, "GNUNET-VPN" might know that the IP-version and transport protocol specifiers have only a few bits of information, but that the IP address range is likely to cause significant branching in the NFA. Thus, a padding length would be specified to ensure that the first k characters of a policy end after the first 4–8 bits of the IP network, resulting in a reasonably large number of initial states. In summary, padding solves the problem allowing diverse applications with their specific regular

expression structures to control the upper bound of synthetic initial states (despite k being a global constant).

3.5 OPTIMIZING THE DFA TO MINIMIZE DHT LOOKUPS

All network operations are expensive and following the transitions stored in the DHT requires a network lookup for each of the states of the NFA. Thus the idea is to decrease the number of required DHT operations for the patrons by optimizing the offerers DFA before storing it in the DHT. Section 2.5 presented four categories of DFA optimization approaches, edge compression, increased stride, path compression and alphabet reduction. We will now discuss the applicability of these optimizations for decentralized evaluation.

3.5.1 *Edge Compression*

Edge compression is useful when the space requirements of a DFA should be reduced. The trade-off for this approach is the increase of required state traversals, for a string of length n that should be matched this optimization results in at most $2n$ state visits. For our approach there is no change in terms of DHT PUT operations, because these are only relevant for states and not for edges. Moreover the number of required state traversals increases with applied edge compression, which counterfeits our goal of optimizing the number of required state traversals.

3.5.2 *Increased Stride*

Increasing the stride of a DFA has the advantage of decreasing the number of required state traversals for string matching. For stride doubling the decrease is $n/2$ for a string of length n. The disadvantage is that the space requirement for storing the additional transitions increases by $|\Sigma|^2$ in theory. Therefore this optimization is a trade-off between faster string matching and storage overhead. Applications where the number of searches by patrons is significantly higher than the number of DFAs published by the offerers can benefit from increased stride. Given that self-loops do not require network operations, our implementation deviates from the standard approach by not adding strides for self-loops.

3.5.3 *Path Compression*

Path compression is similar to stride doubling, as it introduces edges that have a string as their label. However path compression strictly reduces the number of states and state transitions, by replacing linear

paths in the DFA with a single transition and removing intermediate states. Applying maximal path compression will result in an optimal DFA. However as we are merging multiple DFAs in the DHT to an NFA, applying maximal path compression to each DFA will not merge to an ideal NFA in the DHT. For example, consider the following three policies:

$$GNUNET-VPN00010000-V4TCP11000000000000000000000010(0|1)^*$$
$$GNUNET-VPN00010000-V4TCP11000000000000000000000011(0|1)^*$$
$$GNUNET-VPN00010000-V4TCP11000001000000000000000011(0|1)^*$$

They describe the $GNUNET-VPN$ application in version 1 and accept different IPv4 prefixes for TCP connections. For $k = 24$ beeing the initial key length and with maximal path compression, Figure 3.8 illustrates the merging of these policies in the DHT.

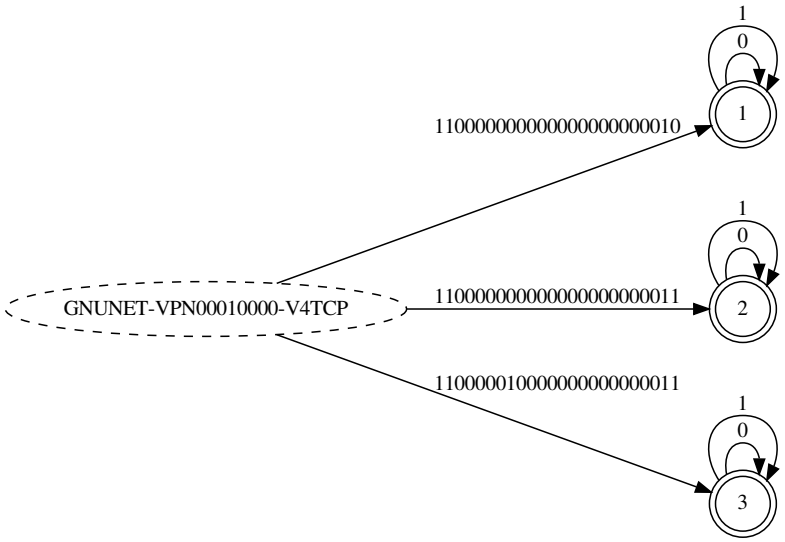


Figure 3.8: Merging of DFAs from the same application where maximal path compression was applied. The stride length is minimal, so that reaching a start state in the DFA only requires one state traversal. The downside is that only the starting states could be merged and the branching at this state is increased.

This example illustrates that maximal path compression does not benefit the resulting NFA in the DHT, as the number of outgoing edges at the $GNUNET-VPN00010000-V4TCP$ state will potentially explode to 2^{24} transitions. Hence in general it is necessary to limit the path compression length. For example with the $GNUNET-VPN$ application for IPv4 prefixes, limiting the path compression length to 8 bits would be a reasonable trade-off between the number of re-

quired state traversals and the possible merging of states in the DHT. Figure 3.9 illustrates the resulting NFA for a limited path compression of 8 bits.

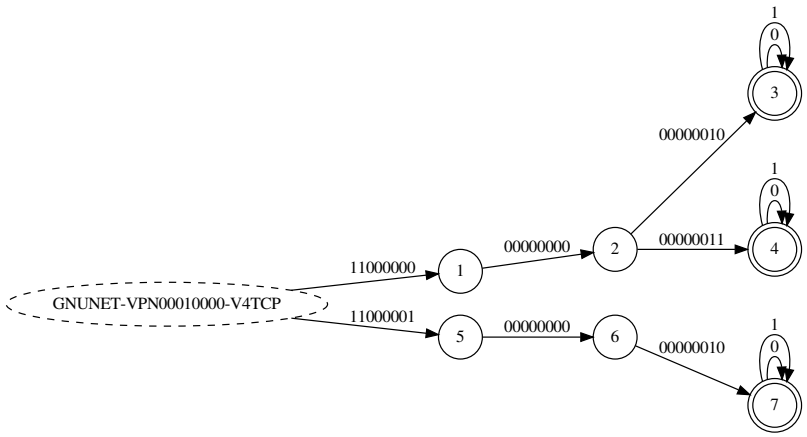


Figure 3.9: Merging of DFA states with a path compression length of 8. The starting state and states 1 and 2 have been merged and the branching at the starting state decreased.

Our implementation allows the application to choose a path compression length, where any value larger or equal to the longest path in the DFA will result in maximum path compression.

3.5.4 *Alphabet Reduction*

The strategy of alphabet reduction groups indistinguishable sets of transitions, replaces them with a new transition and stores this mapping in a table. This optimization is implicitly included in our approach, as all edges of a state are stored under the same entry in the DHT. Each PUT operation stores a value under a certain key. We use the hash of the state identification as the key. For the value we store mappings from sets of labels to DHT keys. Thus our alphabet is already reduced in the sense of DFA alphabet reduction optimization, since only one transition, namely a set of labels mapped to a DHT key, for each next state is stored at each state.

4 IMPLEMENTATION

In this chapter the implementation of our approach to distributed regex evaluation is described. First the overall architecture is presented, followed by the description of the relevant services and the interaction between the individual components. This is followed by a detailed description of the implemented Regex library. Finally, as in this thesis we focused on the problem of finding users that are willing to provide network access using a specified protocol to a particular destination, we describe the Protocol Translation (PT) application that uses the distributed regular expression evaluation as a solution to this problem.

4.1 ARCHITECTURE

The implementation of this thesis' approach is based on GNU's Framework for Secure Peer-to-Peer Networking (GNUnet) [Gro12]. The GNUnet system consists of services, daemons, libraries and user interfaces that are combined into a layered peer instance. Thus the implementation of our system is also organized in a layered architecture consisting of encapsulated components that communicate through APIs. The relevant components are the DHT and Mesh services, the Regex library as well as an application that uses the services. These components are organized in the following layers.

The DHT service builds the foundation layer. On top of the DHT is a layer consisting of the Mesh service and the Regex library. The Mesh service uses the DHT and Regex library. The topmost layer represents the application that communicates with the Mesh service API to integrate regular expression based announcing and discovery of services into a concrete application.

Figure 4.1 illustrates this overall architecture, how the individual components are connected and shows a simplified version of the relevant APIs. Note that this overview does not show all involved GNUnet services, such as Transport (responsible for low level P2P packet transmission), as they are not directly relevant for the architecture overview.

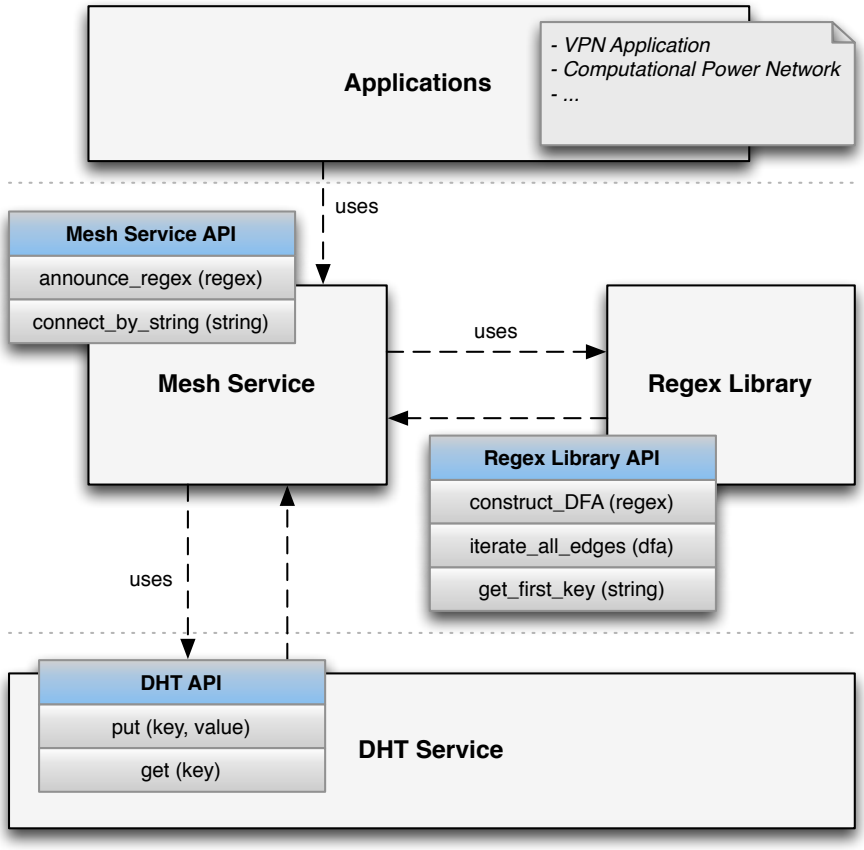


Figure 4.1: Architecture overview diagram with API descriptions for the Mesh Service, Regex Library and DHT Service. The Applications are build on top of the system and use the Mesh Service to announce regular expressions and find peers that match a string.

As illustrated in the architecture overview, the GNUnet services directly relevant to our implementation are the Mesh and DHT service. The following paragraphs give a brief overview of these two services.

MESH SERVICE In GNUnet, the Mesh service is used to establish tunnels to distant peers in the overlay network. For our implementation the Mesh service is extended to provide functionality for announcing regular expressions and find peers matching a search string. Thus it mediates between the applications and the DHT service, as well as uses the Regex library to create DFAs. Apart from abstracting the interaction with the DHT, it also takes care of repeatedly announcing regular expressions as long as the application that is using the service is connected.

DHT SERVICE GNUnet's DHT service implements the Randomized Recursive Routing for Restricted-Route Networks (R^5N) routing algorithm [EG11#2] and provides a simple PUT/GET API. The recursive routing is non-deterministic for the PUT and GET operations

as each time a different routing path is taken with high probability. This results in (unpredictable) latency. Furthermore the GNUnet DHT provides content validation to ensure integrity of the content stored in the system. This is done by providing the application that uses the DHT with hooks to detect malformed key-value pairs. Bad key-value pairs are not stored or forwarded by non-malicious peers.

4.1.1 *Component Interaction*

The following description provides a detailed look at the interaction between the four components shown in the architecture overview. Consider two peers that run an application *app*. The first peer *A* is the offerer and the second peer *B* acts as the patron. The component interaction is separated into announcing — storing a regular expression as a NFA in the DHT — and discovery — traversing the NFA with a search string.

4.1.1.1 *Announcing*

Using *app*, *A* specifies a policy that describes the offered service. *app* then connects to the Mesh service and instructs it to announce the policy. The Mesh service asks the Regex library to construct a DFA for the given policy. The Regex library creates the DFA and returns it to the Mesh service. For each state of the DFA the Mesh service uses the DHT service to store the state in the DHT. At the accepting states, Mesh stores the identification of peer *A*. The announcing of the policy is now completed.

4.1.1.2 *Discovery*

Peer *B*, that is running the same application *app*, is searching for a specific service. Using *app*, it defines a string describing that service. The application then asks the Mesh service for peers that match the string. To get the first DHT key, Mesh uses Regex to get the first key and the number of characters that were consumed. Mesh then does a GET for this key and matches the remaining string against the set of labels. The NFA stored in the DHT is traversed until the string is completely matched and an accepting state is reached. Mesh then returns the identification of peer *A* to the application of peer *B*.

4.2 REGULAR EXPRESSION LIBRARY

The Regex library implements the approach presented in Chapter 3. It is used to construct optimized DFAs for a given regular expression. It uses a graph representation of the automatons — NFAs and DFAs — it constructs, where each automaton consists of a list of states which itself manage a list of transitions.

4.2.1 *API Description*

The Regex library provides an API — which is already briefly shown in Figure 4.1 — that is described in this section. Note that details like type information are omitted.

- `construct_dfa(regex, max_path_len)` - This function is called by the Mesh service to construct a DFA for a regular expression that should be announced. It takes a regular expression string and an maximum path compression length integer and returns a DFA representation. It then constructs a DFA representation. This is done by first creating an NFA (see Section 2.4.1) and then converting it to a DFA (see Section 2.4.2). After that it converts the DFA into its canonical form (see Section 2.4.3) and creates state identification using Algorithm 4. Finally it compresses the DFA paths using Algorithm 3 with the given maximum path compression length.

- `iterate_all_edges(DFA, iterator)` - The Mesh service calls this function to iterate over all edges of the given DFA, including all the synthetic start states. The function calls `iterator` for each state, passing the outgoing edges, which have labels and target states, the state identification and whether or not it is an accepting state.

- `get_first_key(string)` - This function takes a string, which is a search string that should be matched against the NFA stored in the DHT. The function returns the hash of the synthetic starting state. This is done by hashing the first k characters of the string. k is a global constant of the system.

The following additional functions are used for debugging, testing and creation of regular expressions.

- `eval(DFA, string)` - This function takes a DFA representation and a string and matches the string. It returns `true` if the `string` is accepted by the `DFA` and `false` otherwise. This API call is used for testing purposes, which are described in the next chapter.

- `ipv4toregex (ip,netmask)` and `ipv6toregex(ip6,prefixlen)` - These two functions take an ip address (IPv4 in dotted decimal, Internet Protocol version 6 (IPv6) in colon hexadecimal representation) and a netmask or prefix length and return a regular expression. For example, for the IPv4/netmask pair 192.0.2.0/255.255.255.0 it would return the regular expression $110000000000000000000010(0|1)^*$. This is used in the PT application, described in Section 4.3.

- `automaton_save_graph (automaton, options)` - This function creates a graph in the dot [Gra12] format for the given automaton. The options allow coloring of the graph's Strongly Connected Components (SCCs) and adding verbosity. Most of the figures containing automatons in this thesis were created with this API.

The next section describes how the components and the API of the implemented library are tested.

4.2.2 *Testing*

In order to test the Regex library several tests have been implemented. As a basis for the tests sets of predefined regular expressions as well as randomly generated regular expressions and strings are used. The tests are structured into the following parts.

First, to test the correctness of the NFA and DFA construction from regular expressions, we use an evaluation test. This test creates an NFA and a DFA representation using the Regex library and compiles the regex using the GNU C Regular Expression Library [GNU12]. These three representations are then matched against a set of strings and their output (accepting or not accepting) is compared against each other. For the set of predefined static regular expressions these results are also compared to a predefined expected result. The test succeeds if all three give the same output for all strings and regular expressions.

Next, the correctness of state id computation, which implements Algorithm 4, is tested. For a set static set of equal regular expression pairs, each regular expression is used to construct a DFA, which includes computation of the canonical regular expression, which is the union of all regular expressions of the accepting states. The canonical regular expressions of each pair of equal regular expressions are then string compared for equality. Furthermore, for a set of random regular expressions and their corresponding DFAs the canonical regular expression of each DFA is compared to the canonical regular expression of the DFA that is constructed using the original canonical regular expression.

Moreover, the edge iteration API call that iterates over all states and edges of the constructed DFA, including the synthetic start states (see Section 3.4) is tested. For that, a set of DFAs, that are constructed from static regular expressions with sets of expected synthetic start states for each of them, is iterated. For each of the DFAs we check if the defined expected start states are included in the iteration, if the number of iterated transitions is larger or equal to the number of transitions in the DFA and assert that there are no duplicate initial transitions. Additionally, this test includes the option to save the resulting graph in the dot [Gra12] format.

In addition to these tests there are two small tests that check the correctness of IPv4 and IPv6 to regular expression conversion and the graph export function.

4.3 GNUNET PROTOCOL TRANSLATION (PT) APPLICATION

The GNUnet PT / Virtual Public Network (VPN) application implements discovery of peers that are willing to provide network access using a specified protocol to a particular destination. It is composed of the VPN service and the Exit daemon.

The Exit daemon is a program that can be configured to share the Internet connection of the peer. It provides three types of exit functions, Domain Name System (DNS), which allows other peers to use the DNS resolver and IPv4, IPv6 Internet connection access.

The VPN service is the counterpart to the Exit daemon, which provides peers with the ability to route traffic via a virtual network interface to a peer that is running an Exit node. It uses DNS-ALG [Sri12] to hijack the hosts IP traffic. Then, each time an application tries to resolve a hostname, for example when using a web browser to visit *gnunet.org*, the PT application intercepts the request and replaces the reply with an address in the range of the virtual network interface. All traffic received on the virtual network interface is then forwarded to the original destination.

Moreover it is possible to run an Exit daemon and the VPN service on the same host which can be useful for protocol translation between IPv4 and IPv6.

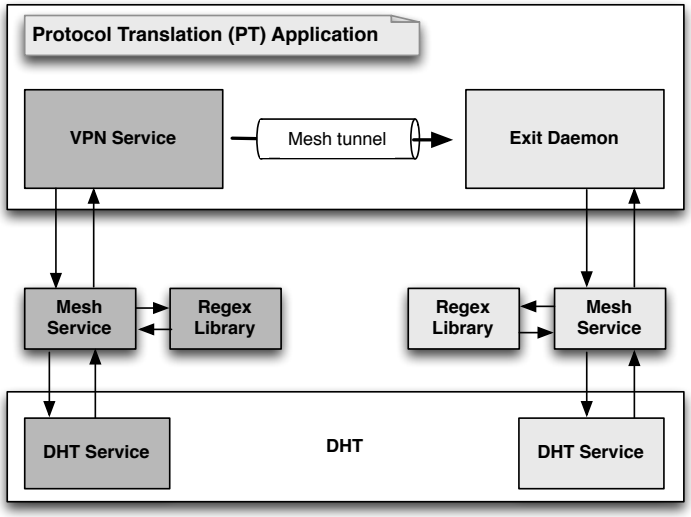


Figure 4.2: Protocol Translation (PT) application overview diagram for two peers, one acting as offerer (highlighted with light grey) that is running the exit daemon and one acting as patron (highlighted with dark grey) that is requesting an exit using the VPN service.

In order to discover peers that are willing to share their Internet connection with a GNUnet Exit daemon, the Mesh service in conjunction with the developed Regex library is used. Figure 4.2 shows an overview diagram for this application.

In the current implementation this system supports differentiation between IPv4 and IPv6 exit nodes for a specified subnet and prefix length, the policy format is defined as follows:

$$GNEX0001PADPADPA\ (4|6)\ \text{<IP-BITSTRING>}$$

The first part contains the GNUnet Exit application identifier $GNEX$, followed by a version string 0001 and padding. This part is followed by 4 or 6, which indicates an IPv4 or IPv6 address respectively. The last part is the actual IP bitstring created by the `ipv4toregex` or `ipv6toregex` API.

5 EVALUATION

In this thesis we focus on the problem of finding users that are willing to provide network access using a specified protocol to a particular destination. In practice this means that some peers act as offerers that publish policies describing the network access they are providing and other peers act as patrons that are requesting particular services. For the evaluation we use a set of policies that describe the IPv4 prefixes routed by each Autonomous System (AS) on the Internet. Before describing the conducted experiments, the next section describes this dataset in more detail.

5.1 DATASET

The Cooperative Association for Internet Data Analysis (CAIDA) is an association that collects data sets of the Internet topology at geographically and topologically diverse locations and provides these data sets for the research community. The Routeviews Prefix to AS mappings Dataset (pfx2as) is one of such and contains IPv4 prefix to AS mappings [Coo12], derived from the University of Oregon's Route Views Project [Uni12]. It contains a mapping from IPv4 prefix including the prefix length, to a unique AS ID. This dataset is updated daily and as of this writing contains 440,448 lines with 40,696 unique AS IDs, where some IPv4 prefixes are assigned to more than one AS ID. As already mentioned, for the purpose of this evaluation we need to have a set of policies that describe the IPv4 addresses routed for each AS. Thus the pfx2as data is parsed using a script where the output is a set of files — one for each AS ID — that contain the corresponding policies. The following is an example policy from this dataset:
$GNUNET - VPN - 0001P11000001100101100010100 1(0|1)*$
For the dataset this results in 40,696 files, one for each AS ID containing 451,282 lines in total.

5.2 THEORY

As shown in Section 3.3, announcing each regular expression from the pfx2as dataset and thus publishing the individual DFAs results in a merged NFA. For the used dataset, this merged NFA is expected to have the following properties.

- For the 451,282 lines lines, containing a regular expression in each line, the merged NFA is expected to have at most 451,282 accepting states.

- Predicting the exact number of states in the merged NFA is not possible, as we cannot say in advance how many states will be merged. Furthermore this depends on the chosen path compression length as well as the IPv4 prefix length of the AS policies. Assuming no state can be merged, an average IPv4 prefix length of 20 and disabling path compression, this maximum number of states is bound by $20 * 451,282 = 9,025,640$ states.
- The number of synthetic start states depends on the prefix length. For a prefix length $k = 24$ and a fixed prefix of length 16 the expected number of synthetic start states is 2^8.
- The non-determinism at each state — meaning outgoing edges with the same label to different states — is expected to be low. This is because only states with overlapping policies will have non-deterministic transitions, for example when merging the following two regular expressions: $010101(0|1)*$ and $0101011010(0|1)*$.
- The decrease of states and transitions for increased maximum path compression length is expected to be equal. For this consider a linear path in the DFA with n states (including start and end state of the linear path) and $m = n - 1$ transitions. When replacing the m transitions with one, the start and end state are kept. Thus $n_{new} = 2$ and $m_{new} = 1$ and the decrease of states is $n - 2$ and of transitions $m - 1 = n - 2$.

In the following sections we first conduct a simulation, followed by an emulation in order to evaluate the implemented system and check the above assumptions.

5.3 SIMULATION

Simulating the distributed merging of DFAs into an NFA allows us to analyze the NFA that should result in the DHT without the need for setting up and deploying a P2P network.

5.3.1 *Experimental Setup*

Instead of using the Mesh service to announce regular expressions, this simulation replaces it with a profiler. The profiler substitutes the DHT with a database and uses the Regex library to construct DFAs for each of the policies in the pfx2as data set. All edges of each DFA are inserted into a table. Such a table then represents the merged NFA. Figure 5.1 illustrates this setup.

The database used in this setup is a MySQL database and the schema for the tables is shown in Appendix A.1. Appendix A.2.1 lists the information needed to run the profiler.

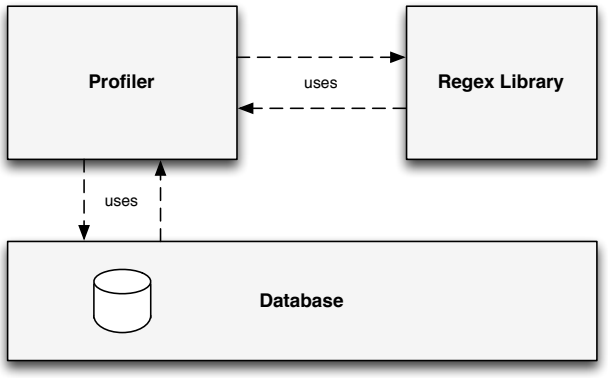


Figure 5.1: Experimental setup for simulation. Compared to the architecture of the implementation illustrated in Figure 4.1, this setup replaces the Mesh service with a profiler and substitutes the DHT with a database. This system provides a way to analyze the resulting merged NFA without having to deploy it to a distributed setup.

5.3.2 *Measurements*

To evaluate different scenarios the simulation has been conducted for different path lengths. We will now present the results of our measurements.

Figure 5.2 shows the number of transitions and states for different path compression lengths.

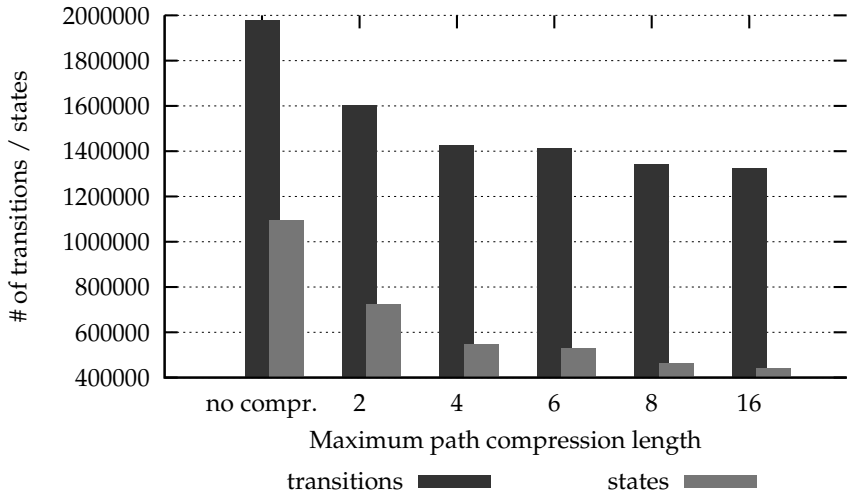


Figure 5.2: Number of transitions and states in the merged NFA for different path lengths.

As expected, the number of states and transitions decreases with increased path compression and the absolute number of decreased states and transitions are equal. A simple path compression of length two already decreases the number of required states from 1,095,441 to 723,398. Moreover there is another significant change of required

states and transitions from maximum path length 2 to 4 and 6 to 8. These observations can be explained by the class-based routing scheme and the dotted decimal format used for IPv4 addresses. Moreover, when choosing an optimal maximum path length for deploying this application this figure suggests that a longer path length results in fewer states and transitions and therefore the optimum would be maximum path compression. However, studying the number of outgoing transitions at each state is another important criterion that has to be taken into account.

Figure 5.3 shows — for different path lengths — the number of outgoing transitions on states in the NFA graph and how many states have this particular number of outgoing transitions. Note that this graph contains only information for maximum path lengths 1,2 and 4, as the number of outgoing transitions increases to a high value.

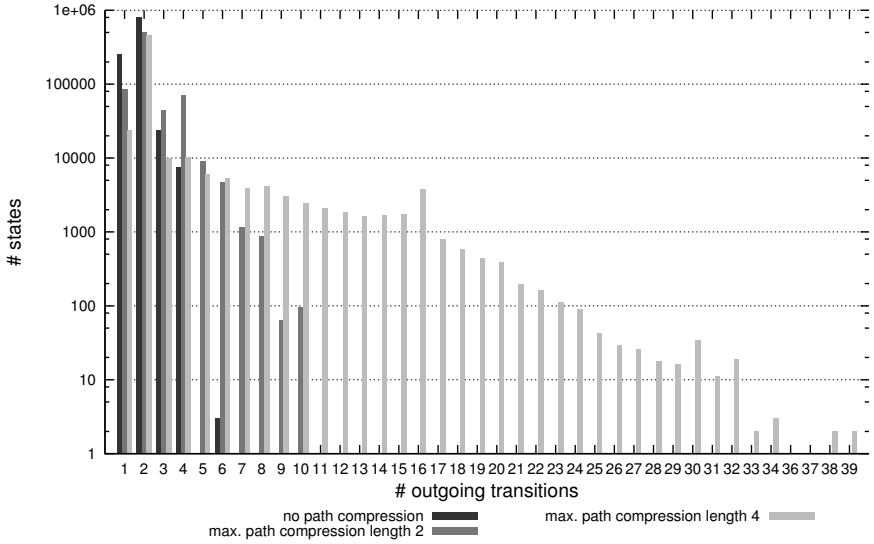


Figure 5.3: Count of states in the merged NFA grouped by their outgoing transitions, for maximum path lengths $1, 2, 4$.

In Figure 5.4 we show the Complementary Cumulative Distribution Function (CCDF) for maximum path lengths 6, 8 and 16. It illustrates that for longer path lengths the number of outgoing edges at a few percentage of the states grows quickly. For a path length of 16 there exists a state that has $12,645$ outgoing transitions.

When observing the values for disabled path compression it is remarkable that there exist states that have more than two outgoing transitions. Instead there is a few percentage of states that have 3, 4 and 6 outgoing transitions. This is explained by the non-determinism at some states.

Now we examine the non-determinism distribution among all states. A non-determinism at a state in the merged NFA indicates the number of outgoing transitions that have the same label and thus there exist several paths in the NFA that need to be traversed when matching a

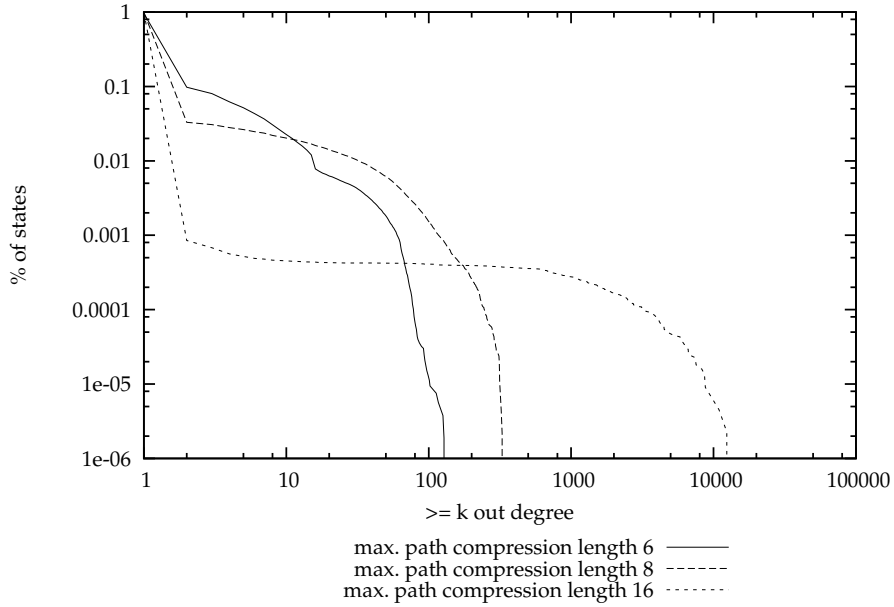


Figure 5.4: Complementary Cumulative Distribution Function of states and their corresponding outgoing transitions, for path lengths 6, 8 and 16

string. Figure 5.5 shows that the maximum number of transitions with the same label at any state in the merged NFA is 3 and only if path compression is disabled. If path compression is enabled the maximum number of non-determinism is 2.

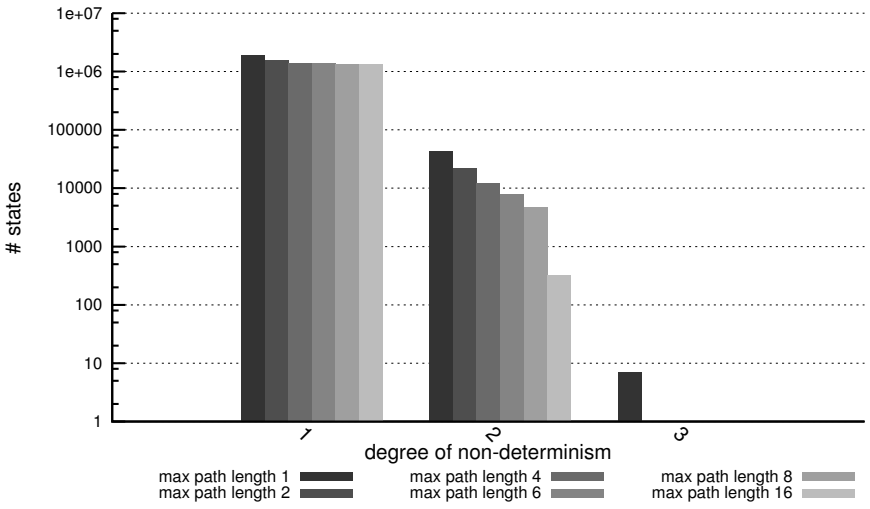


Figure 5.5: Degree of non-nondeterminism at states in the merged NFA

Additionally we examined the state increase in the merged NFA with enabled double striding, for maximum path compression lengths 4, 6 and 8. As expected, the number of states did not change. The number of transitions increased by 38%, 36.88% and 31.78% respectively.

As already discussed at the beginning of this section, there are more IPv4 prefixes then total ASs. The average number of IPv4 prefixes per AS is 11.08. However there exist some ASs, where the number of IPv4 prefixes is larger than 100. Thus creating a regular expression for each prefix and announcing it will produce a high number of duplicate (key, value) pairs at the same peer that is responsible for one AS. This is not desirable as this will generate redundant PUT operations for each duplicate (key, value) pair. For this reason the regular expressions are combined to a union. The example shown in Appendix B.1 shows that combining the prefixes to one large regular expression and converting it to a minimized DFA results in fewer states and transitions. We have conducted a simulation for a subset of 100 ASs with 2778 regular expressions and a maximum path length of 4. As to be expected the number of accepting states decreased from 2778 to 100. Moreover the number of transitions in the merged NFA decreased from 9901 to 2641 and the number of states from 4424 to 1536. While this approach reduces the number required PUT operations for each peer, the NFA non-determinism increases slightly from a maximum of 2 outgoing transitions with the same label at 42 out of 9901 states to a maximum of 3 equal outgoing transitions at 3 out of 2641 states and 2 at 23 states.

5.4 EMULATION

To evaluate the implemented system in an emulated environment, the GNUnet testbed service is used. This is an updated version of the GNUnet testing framework [EG11#1]. It facilitates the setup of a distributed GNUnet deployment and enables us to start several peers on several different hosts.

5.4.1 *Experimental Setup*

The experimental setup for this evaluation uses the CAIDA database for the pfx2as data, as described above. Each AS ID corresponds to a peer, which is then responsible for announcing the policies of this AS. The InfiniBand cluster at LRR-TUM [TUM12] is used to execute a distributed deployment of the implemented system. The cluster contains 25 nodes, with a 4x Opteron 850 2,4 GHz core and 8 GB memory each, that are interconnected via an InfiniBand 4x switch. Figure 5.6 illustrates the testbed setup for this evaluation. A master controller is created on one dedicated host which then connects to slave controllers, one for each host machine. On each host a number of peers are started and the started peers on the slaves are randomly linked. Each of the peers is responsible for announcing its corresponding regular expressions. This is done when a peer is started on the slave controllers. A simple daemon is started automatically that connects to the peer's Mesh service and announces the corresponding regular

expression read from a file. In Appendix B.1 we show an example regular expression and the corresponding generated DFA for an AS with a long list of IPv4 prefixes.

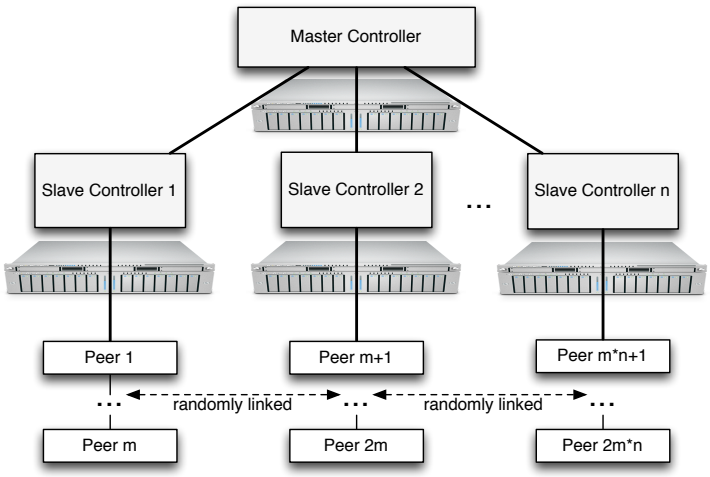


Figure 5.6: Illustration of the InfiniBand testbed setup where the master controller is started on the first host, which then connects to the slave hosts where a number of peers are started. All started peers are then randomly linked.

In addition to the LRR-TUM [TUM12] InfiniBand cluster, a set of experiments have also been conducted successfully on single hosts. For this the one master testbed controller and one slave controller as well as all the peers are started on one machine.

Searching of strings is triggered from the profiler running on the master controller. After all links between peers have been established, the profiler reads a set of strings from a file and for each file connects to the mesh service of a different peer and starts the string search. Appendix A.2.2 lists the information needed to run the profiler.

5.4.2 *Measurements*

In this section we present the results of our measurements. The evaluation has been conducted for four different testbed scales, 100, 500, 1,000 and 5,000 peers.

In Table 5.1 we compare the statistic for the differently scaled experiments from 100 to 1,000 peers. Each of the experiments has been repeated five times and for each run the search strings have been randomly sorted. This ensures, that the chance of one peer matching its own regular expressions is low. The number of random links between the peers has been set to *number of peers*$\times$10. Note that the absolute values cannot be directly compared in this evaluation, as each test was run on a different system.

The results for the regular expression statistics show, that our approach scales well for the conducted evaluations, as the number of

Average	100 peers	500 peers	1,000 peers
kilobytes PUT	70.71	5,095.94	7,437.57
kilobytes GET	55.85	556.29	895.92
regex DHT blocks iterated	9.35	9.62	9.59
regex nodes traversed	4.78	4.28	4.65
regex edges iterated	27.49	27.52	28.27
peers connected	28.66	43.21	48.44
search duration	12.48 s	13.28 s	101.94 s

Table 5.1: Average statistics per peer for emulations with 100, 500 and 1000 peers.

iterated edges, the number of traversed nodes and iterated mesh blocks did not change significantly, although the number of peers was increased. Note that the number of iterated edges is much higher than the traversed nodes. This is due to the fact that each edge of a in the value of a (key, value) pair stored in the DHT is iterated and because the DFA, as shown in Figure B.1, has two closures at the accepting state.

Figure 5.7 and Figure 5.8 illustrate the search duration for the runs with 100 peers and 500 peers accordingly.

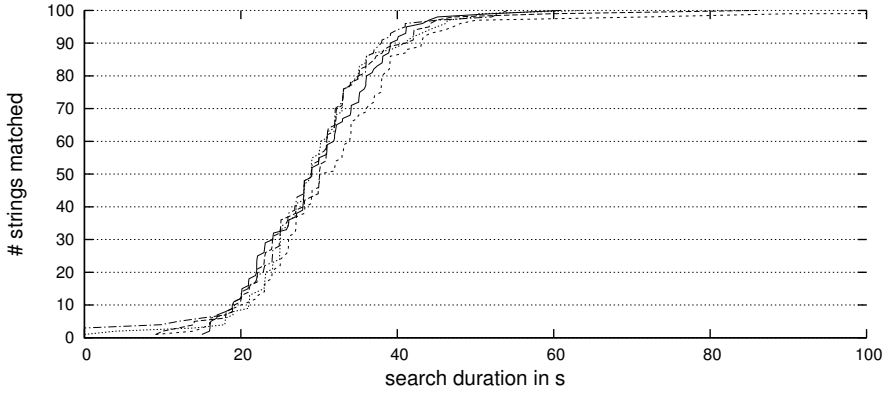


Figure 5.7: Search duration for five runs with 100 randomly connected peers, 100 regular expressions and 100 search strings.

Although the absolute values for search duration cannot be compared directly, as explained above, both figures show the same pattern. The majority of the strings is found after the same amount of time. Some strings are found quickly, which can be explained with the fact that some strings are searched from the same peer that stored a matching regular expression. Furthermore it might happen that a direct neighbor has announced a matching regular expression, in which case it will also match fast. Finally some strings take a long time to match or even fail to match after a certain timeout. This can be explained

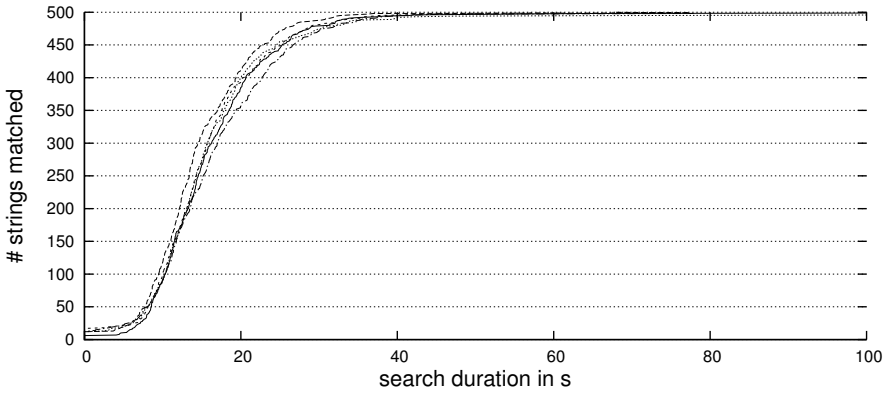


Figure 5.8: Search duration for five runs with 500 randomly connected peers, 500 regular expressions and 500 search strings.

with the unpredictable latency of the underlying GNUnet DHT, which performs randomized routing.

In addition to the above measurements, we conducted an experiment in the cluster with 5,000 peers connected with 10,000 random links and a maximum path compression length of 4, 8 and 16. The string matching was run for 20 minutes. Table 5.2 shows the average statistics for the three maximum path compression lengths.

Average for max. path len:	4	8	16
kilobytes PUT	2,316.32	1,726.67	1,358.97
kilobytes GET	747.34	650.74	591.12
regex DHT blocks iterated	11.93	9.85	9.72
regex nodes traversed	6.85	5.43	4.61
regex edges iterated	39.33	29.61	28.65
peers connected	36.52	38.75	40.06
search duration	232.30 s	183.52 s	164.14 s

Table 5.2: Comparison of maximum path length 4, 8 and 16 for 5000 peers.

As already discussed in the simulation, we notice a significant decrease of average required edge iterations and node traversals from maximum path compression length 4 to 8. Furthermore the average search duration decreases significantly for longer path compression and increased average peer connections. Furthermore, for this moderate-sized testbed, the average number of bytes that have been consumed for PUT and GET operations is acceptably.

Figure 5.9 illustrates the change in search duration for the three different maximum path compression lengths.

This illustration shows that not only the average search duration is performing better for increased path compression, but also the

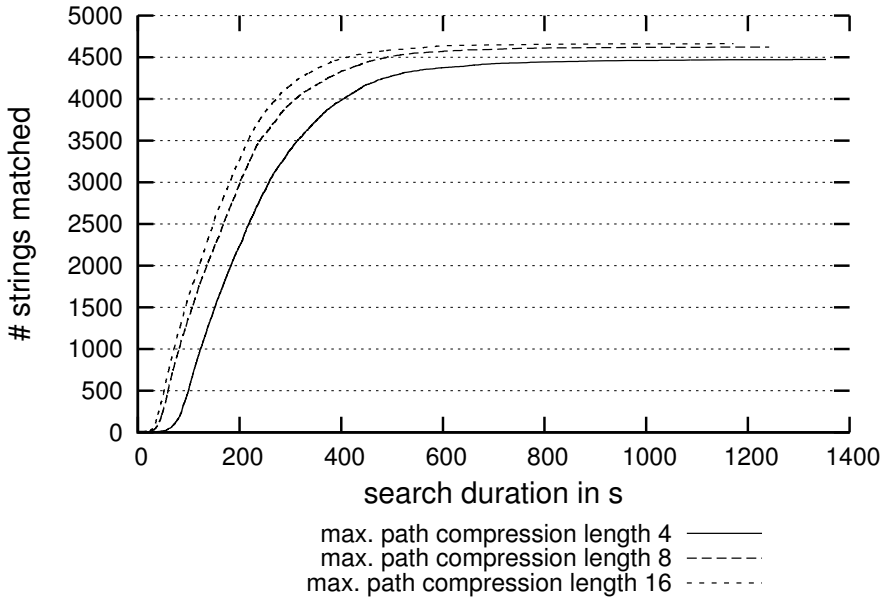


Figure 5.9: Search duration for 5000 peers and maximum path compressions 4, 8 and 16.

absolute number of successfully matched strings in the time frame of the evaluation is higher.

5.5 SUMMARY

Our evaluations show that merging of the DFAs using the independent feasible merge described in Section 3.3 to an NFA is feasible and performs well for our scenario. The simulation shows that the resulting NFA has the expected properties and the non-determinism of transitions at states is low for the evaluated dataset. The emulation shows that our approach is actually viable for a deployment in a moderate-size testbed and suitable for applications that can tolerate moderate latency.

6 RELATED WORK

The work related to our presented approach is structured into three different categories. Section 6.1 contains approaches that build a structure on top of an existing DHT, which is what our approach is also based on. Section 6.2 discusses approaches that construct a new P2P overlay and Section 6.3 presents works that follow other methods to provide advanced queries on structured P2P overlays.

6.1 STRUCTURE ON TOP OF DHT

This section presents selected approaches — *Data Indexing in Peer-to-Peer DHT Networks*, *PastryStrings* and *Prefix Hash Tree (PHT)* — that provide support for advanced queries in DHT-based overlays by building a structure on top of an existing DHT.

6.1.1 *Data Indexing in Peer-to-Peer DHT Networks*

Garcés-Erice et al. propose a system [Gar+04] to discover objects stored in a DHT. A loose information structure is build on top of a DHT by indexing the objects under a trail of their associated keywords.

In this approach the objects are indexed under their descriptor's hash. The descriptor consists of the object's keywords that are structured in an XML format. File descriptors can contain arbitrary information about the file, for example the filename, the creation date, the file size and the author. When indexing a file, an arbitrary number of sets of keywords, that can be freely chosen by the application, are inserted into the DHT with links to more specific keyword sets. Keyword sets have an XML Path Language (XPath)-like syntax that contains information from the files descriptor. Figure 6.1 shows an example taken from [Gar+04] with distributed indexes for three files from a bibliographic database.

Searching for files that are indexed in the DHT is done with a general query that consists of keywords with an XPath-like syntax. The hash of this query is used to follow the previously inserted trail of more specific keyword sets until the most specific set of keywords is found. The search ends at the most specific set of keywords, which is the objects descriptor, as this is where the searched file is stored (or the peer that offers this file).

Building and maintaining the indexes in this approach is arbitrary and left to the application. This makes it possible to speed up popular queries by inserting a direct link from a generic query

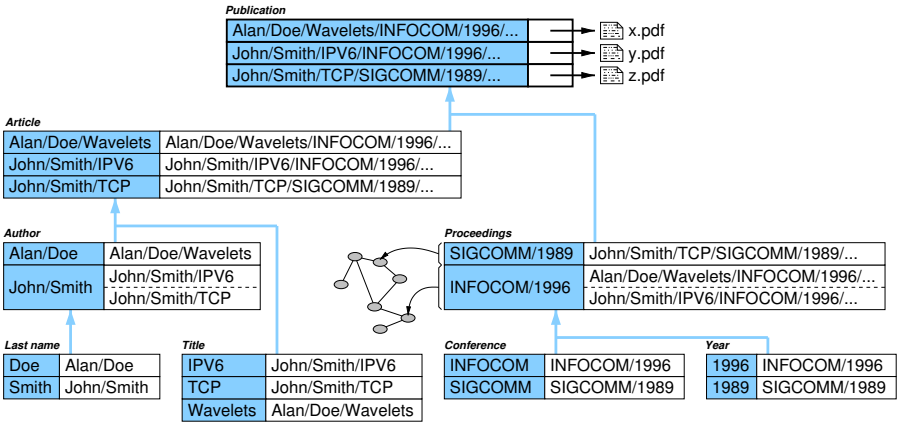


Figure 6.1: Distributed indexes for three documents from a bibliographic database (Illustration from [Gar+04]). The publication index is the most specific query that is used as a the descriptor d, which itself is hashed to derive the key for storing the files in the DHT. For example, a query for the generic query *IPV6* would lead to the more specific query *John/Smith/IPV6* and finally to the file stored under the query *John/Smith/IPV6/INFOCOM/1996/...* (Illustration from [Gar+04]).

directly to the file, for instance adding an entry for *Smith* to *John/Smith/IPV6/INFOCOM/1996/...* in the above example. Performance and usability of this approach heavily depend on the underlying data and query structure.

6.1.2 *PastryStrings*

PastryStrings [ATo6] is an infrastructure on top of a DHT for supporting numerical range and comparison queries, as well as prefix, suffix and containment string matching.

The architechture is based on a tree structure for storing and matching events to subscriptions in the DHT. A forest of dynamically created string trees is build when new requests for storing subscriptions arrive. The root node of each tree corresponds to the first character of the string that should be stored. Storing the root is done by hashing the first character of the string and performing a PUT on the underlying Pastry DHT. Each node of the tree maintains a routing table. The routing table contains a mapping from the next character in the stored string to child nodes.

To find the peers that store a subscription to an event, the tree is recursively traversed. Each peer that stores a matching subscription at any of the traversed leaf-nodes is send a message. Figure 6.2 shows an example for the string-typed event to subscription matching.

PastryStrings supports the "*" prefix and suffix predicate on strings, which is done by storing a subscription for strings ending in "*", for example *abc**, at each child-tree of the node that is responsible for

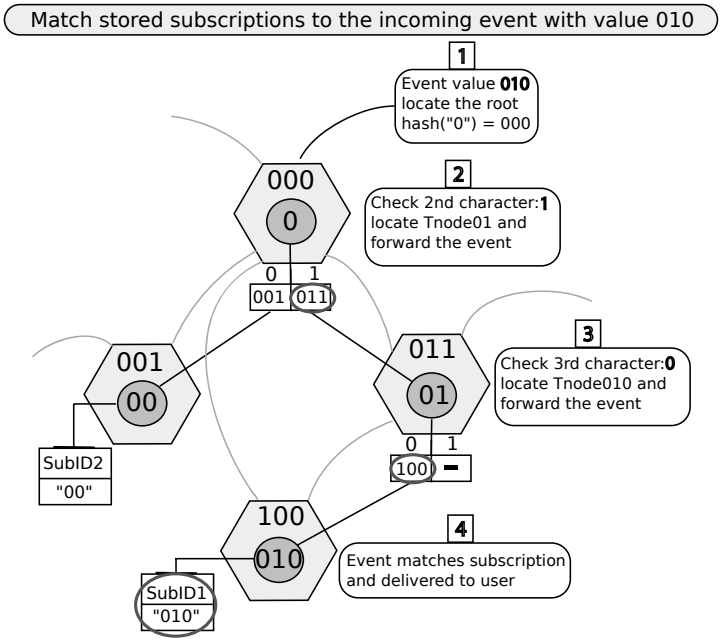


Figure 6.2: PastryStrings string-typed event to subscription matching for the string **010**. The root is found by hashing the first character of the string $hash(\text{"0"}) = 000$ and then the tree is traversed using the routing tables stored at each node, until the matching leaf node is found (Illustration from [AT06]).

storing subscriptions for *abc*. In order to support numerical attributes to subscriptions, sub-ranges of numerical values are assigned to each node. Leaf-nodes just store the value and their parents store the range that includes all of their child-nodes. For example, for the two leaf-nodes *0000* and *0001*, their parent *000* stores the range *[0,2)*. Thus a subscription for the range *[0,2)* is stored at the node responsible for *000*.

To balance the load on popular nodes (i.e. nodes that are close to the root of a tree) the tree forest is replicated so that storage load is partitioned over several nodes. The number of replications is defined by a global replication factor f, which corresponds to the number of replica trees in the DHT. When inserting a new root it is inserted f times and each replica's id contains its number. Lookup is done by randomly choosing a number $r < f$ and hashing it together with the first character, so that the replica tree is randomly chosen.

6.1.3 *Prefix Hash Tree (PHT)*

A PHT is a data structure presented in [RHS03] that is used to build a trie on top of any given DHT implementation in order to support range queries over DHTs.

Every node in a PHT has an associated prefix label. The root of a PHT is labeled with an attribute name defined by the application. For

a node with a label l, its left child node is labeled $l0$ and its right child node is labeled $l1$. Data items are stored at the nodes with the longest common prefix between the nodes label and the data item to be stored. Data items are only stored at the leaf nodes of a PHT. If the parameter of maximum allowed number of data items, that can be stored at each node, is exceeded, the corresponding node is split into child nodes.

Lookup of bitstrings can be directly performed by hashing the string and using the hash as the key for the DHT GET operation. The additional benefit of this approach is the support for range queries. In order to do that the subtree in a PHT with the longest common prefix is traversed to find all the leaf nodes of a specified range.

To avoid overloading of root nodes, not all traversals have to start from the root node. Lookups are executed by hashing possible prefixes of the search keyword and performing a GET for each of these hashes, where the PHT node with the longest common prefix provides the result. Thus for a string $011*$ the prefixes 0110 and 0111 could be used to perform a GET and avoid overloading of the node storing 001.

6.1.4 *Comparison to our approach*

Similar to our approach, the approaches presented in this section all build a certain structure on top of a DHT.

The system by Garcés-Erice et al. is similar to our approach, as both approaches use a string that describes indexed data to find appropriate matches. However Garcés-Erice et al. approach is limited to simple keywords that describe indexed data, where our approach allows indexing with regular expressions.

When comparing our approach to *PastryStrings* the subscriptions can be seen as offering a service using a policy. The publishing part can be seen as a patron, which is searching for a service using a string. We, as well, use the hash of the first characters to find the start of the graph structure contained in the DHT. Both approaches do not allow direct *jumping* to a particular node. However, in contrast to *PastryStrings*, which is limited to range, string prefix and string suffix queries, we support regular expressions, which has more flexibility and is essentially a superset of the mentioned types of queries. Moreover in our approach we avoid overloading of initial peers by using policies which let the application choose a reasonable prefix. In contrast to this, *PastryStrings* replicates the entire tree forest to avoid overloading of peers.

PHT is similar to *PastryStrings*, but allows direct lookup of any node in the tree and uses this to avoid overloading of higher level nodes.

Using these approaches for the discovery of offerers that provide a certain service, the service descriptions are limited to simple range queries or string prefix/suffix matching, as none of the named ap-

proaches provides more advanced query types than the ones mentioned.

6.2 P2P OVERLAY

In this section we present selected approaches — Cubit and DPMS — that build structured P2P overlay networks with support for advanced query matching.

6.2.1 *Cubit*

Cubit [BS08] is a loosely structured overlay network providing approximate keyword matching by mapping objects to keywords, which are then stored at nodes with related keywords.

Each Cubit node organizes its peers into concentric rings (see Figure 6.3), where it retains links to neighbors in each ring. Thus each node has links to nearby, as well as, far away peers. Each node stores the objects to which it is the closest (in the sense of edit distance between keywords). Keywords are derived from the information associated with an object. For example for files it could be the filename plus some tags. The nodes are distributed in the same space as the keywords. Node IDs are randomly chosen from the set of keywords and uniqueness is ensured by detecting ID collisions. Each object is fully described by a set of keywords. When inserting a new object, the object descriptor is replicated at the closest nodes for each of the keywords by first inserting the object at the closest node and this node then performs the replication at its closest neighbors.

Searching for approximate matches to a keyword is provided by looking up the nearby nodes to a given keyword. The search starts at the local peer and iteratively queries nearby nodes close to the keyword, until no nodes can be found that is closer to the keyword.

As stated above, Cubit supports approximate string keyword search. In addition it also supports queries with logic operators. For that searches are converted to a set of conjunction clauses that are connected by OR operators. The standard search described above is performed to find close matches to the positive terms in each clause, as well as to find all negative terms (*NOT*) that then act as a filter. The union of the filtered results is returned.

Load balancing of popular keywords stored at one node is done by creating a virtual copy of the node and storing this copy at its neighbors, so that queries that arrive at the neighbors can be answered without involving the original node.

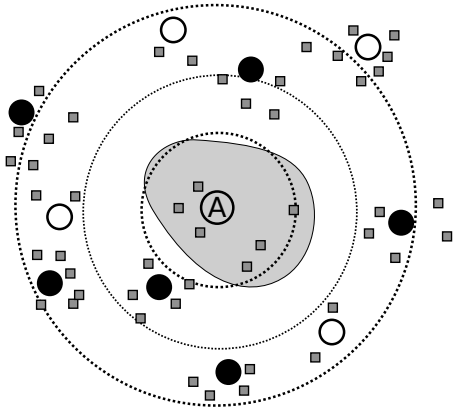


Figure 6.3: Illustration of a cubit node A that shows the concentric rings into which the node organizes its peers. Each concentric ring stands for a specific keyword distance. The outer rings have a bigger keyword distance than the inner rings. The solid circles are the peers to which A retains connections, empty circles are other nodes. The squares represent object keywords in the system. Each keyword in the shaded region is in the keyword subspace where A is the closest node.

6.2.2 *DPMS*

The Distributed Pattern Matching System (DPMS) [ABo6] is a structured P2P network that supports string pattern matching based on Bloom filters.

In DPMS each object is indexed using a bitstring. For each keyword to an object (for example files) a bit in the string is set to 1. To index a file with several keywords, the keywords are split into n-grams and then encoded in a single bitstring. To efficiently encode a set of keywords to a bitstring a Bloom filter is used. Each peer advertises its bitstring index. The advertised patterns are organized in a latice-like hierarchy. Peers have two roles, they act as offerers that advertise their indices and/or as indexing peers. Indexing peers are positioned above the lowest level in the lattice hierarchy and aggregate multiple indices from lower level peers. They help in routing requests to the offerers. [ABo6] suggest a special aggregation scheme that aggregates several index bitstrings into one: if two bitstrings have different bits at the same position, the bit in the aggregated bitstring is set to X, which indicates that at this position both 0 or 1 are possible values.

Querying for indexed content is done by splitting the search keyword into n-grams and encoding it to a bitstring using a Bloom filter. Matching to advertised bitstrings is done by comparing the 1-bits of the search query to the advertised bitstrings. For example the query bitstring 0**1**0**1** would find a match in the advertised bitstrings 0**1**1**1** and 1**1**0**1**, but not 1011. The query is then routed by first checking the local information and if no match was found, forwarding the request to the higher level indexing peers until a aggregate match is found.

The peer with a matching aggregate bitstring then forwards the query down in the hierarchy until an offerer that advertises a matching index is found.

DPMS supports Bloom filter based partial- and multi-keyword search. For example a offerer might index the keywords "invisible man" and another offerer the keywords "visible woman", these are split into tri-grams "inv, nvi, vis, isi, sib, ibl, ble" and "vis, isi, sib, ibl, ble, wom, oma, man". Searching for the partial keyword "*visi*man*", which is split into the tri-grams "vis, isi, man", will find both advertised keywords.

An additional optimization is presented to minimize the number of *X*-bits in the aggregated Bloom filter bitstrings — which is an NP-complete problem — by using a heuristic approach.

6.2.3 *Comparison to our approach*

When comparing our approach to the approaches presented in this section, the main difference is, that both systems construct a new overlay P2P network, where we build on top of an existing DHT. In contrast to our approach, where indexing is done with regular expressions, both *Cubit* and *DPMS* use keywords consisting of plain strings to index data and then use more expressive queries to match a number of stored keywords. This does not fit to our approach where a specific string is searched and the offerer stores the data structure (DFA) in the DHT. Thus using these approaches is generally not feasible for our scenario.

Apart from this major limitation these approaches could be applied to our scenario. The approximate matching and boolean search facilities in *Cubit* could be used to find matching services, given that all possible service descriptions are indexed using keywords. Service discovery with *DPMS* could be achieved by constructing service descriptions that benefit from the splitting into n-grams. Nevertheless neither of these approaches supports the flexibility provided by general purpose regular expressions.

6.3 OTHER

This section presents two approaches that use other methods to provide advanced queries on structured overlays. The work by Karnstedt et al. — *Similarity Queries on Structured Data in Structured Overlays* — provides support for databases on DHT-based overlays and *Cooperative Information Sharing System (CISS)* alters the hash function of the underlying DHT to allow range queries.

6.3.1 *Similarity Queries on Structured Data in Structured Overlays*

Karnstedt et al. present an approach [Kar+06] to store and select entries of relational databases on DHT-based overlays. The approach allows database-like similarity queries.

In a classic relational model the relations $R(A_1, \ldots, A_n)$, where A_i represents the attribute name (or column name), with tuples of values $(id, v_1, \ldots, v_n)$ would be stored in a horizontal organization, for example a table in an SQL database. This organization requires that all users agree on a common database schema. This is not feasible, as the motivation for supporting similarity queries on DHT-based overlays in this work is grounded on public management of structured data (e.g. LDAP). The presented approach organizes the relation vertically by indexing triples of the form $(id, A_1, v_1), \ldots, (id, A_n, v_n)$ in the DHT. Indexing is done by hashing different parts of the triples and using this hash as the key to store the triples in the DHT. To support object lookups each triple is hashed using the respective *id*. Supporting selections on the stored data that are of the form $A_i \geq v$, for example $price \geq 50$, is done by hashing each triple under $A_i \# v_i$ (# denotes concatenation). And finally hashing on v_i allows for keyword-like queries like $any\ attribute = v$, for example $price = 100$. This storage strategy causes an overhead in storing space, but allows for relational queries on the stored data without the need for a fixed schema.

The system is based on the P-Grid DHT [Abe+03], which uses a binary trie structure to organize each peers routing table. P-Grid allows for exact and sub-string search, as well as range queries on keys, which is necessary for querying the data indexed in the DHT by using a relational query language that supports *SELECT* queries with *WHERE*, *ORDER BY*, *LIMIT* and *OFFSET* statements. For example consider the following query:

Select name, horsepower (hp) and price of the 5 most powered cars below a price of 50000 from a collection of cars, represented in a database by typical attributes such as name, mileage, price, etc. which is expressed as:

```
SELECT ?n,?h,?p
WHERE { (?o,name,?n) (?o,hp,?h) (?o,price,?p)
FILTER (?p < 50000) }
ORDER BY ?h DESC LIMIT 5
```

Moreover, similarity queries, which return results that are similar to a given search string, are proposed. These are based on inserting q-grams for each $A_i \in R$.

6.3.2 *Cooperative Information Sharing System*

CISS ([Lee+07]) mediates between P2P applications and an existing DHT. Instead of using a hash function for mapping objects to DHT

keys, which facilitates de-clustering and is adverse to advanced access operations (such as range queries), CISS uses a locality preserving function to achieve a high level of clustering.

The locality preserving function defines two different key encodings that preserve clustering. Numerical types are simply rescaled and attribute clustering is preserved, because the numeric type is a total ordered set. To support clustering of strings, the strings have to be hierachically ordered by the application. For example consider strings that describe locations, which can be ordered by generality: "Europe, Germany, Bavaria, Munich". A prefix is added to each string by the locality preserving function, to preserve the clustering.

Efficient range queries are suppported by clustering semantically related objects, which allows searching for multiple key-values by a single DHT lookup. A forwarding-based routing protocol for queries is used to route multi-dimensional range queries. This is done by only looking up the nodes that correspond to the first key of each cluster. The query is then forwarded to succeeding nodes until all relevant objects are retrieved. Relevant objects are directly returned to the searching peer.

In addition to multi-dimensional range queries, CISS also supports continuous updates, because semantically related objects are clustered, continuous updates can be routed to the same node without performing additional DHT operations.

6.3.3 *Comparison to our approach*

The only similarity of the presented works in comparison to our approach and with respect to the underlying structure is the usage of a DHT. Applying CISS to our scenario — discovery of peers that offer a certain service — would mean that an offerer would have to explicitly insert strings that fully describe the offered service. Similar services would then be clustered and the patron can discover services using range queries. The insertion of all strings that describe a service would only be feasible for a small number of strings. The approach of Karnstedt et al. could be applied to our scenario by mapping policies to database tables that can be stored in the DHT. This requires storing each capability description under several entries in the table, for example each IPv4 address for which the peer provides an exit. Searching for capabilities would then be done using SQL-like queries and. Thus, although it would be possible to use their approach, our approach requires less storage space (one regular expression in contrast to several table entries) and searching for a particular service can performed using simple strings instead of complex queries.

7 CONCLUSION AND FUTURE WORK

This thesis describes a method to search a DHT-based overlay network using a string that matches an advertised regular expression. The communication complexity of the approach depends heavily on the specifics of the DFAs and the implementation of the DHT. The expected latency is linear in the length of the search string and generally suitable for applications that can tolerate moderate latency.

In future work, the performance of the new method should be compared to other approaches to distributed search. We also hope that the new expressive search method will support the development of new applications which were previously difficult to implement due to the lack of an efficient and expressive search mechanism.

Furthermore, as the method only covers searching using a string, an open problem is searching using a regular expression for an advertised string. Finally, our experimental evaluation was performed using a moderate-size testbed; in the future, experiments using a testbed with a more realistic number of peers and a broader range of applications might provide additional insights.

BIBLIOGRAPHY

[ABo6] R. Ahmed and R. Boutaba. "Distributed Pattern Matching: A Key to Flexible and Efficient P2P Search". In: *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*. Vancouver, BC, Canada: IEEE, 2006, pp. 198–208.

[Abe+03] K. Aberer et al. "P-Grid: a self-organizing structured P2P system". In: *SIGMOD Rec.* 32.3 (Sept. 2003), pp. 29–33.

[Aho+06] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2006.

[App97] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Dec. 1997, pp. 27–30.

[ATo6] I. Aekaterinidis and P. Triantafillou. "PastryStrings: A Comprehensive Content-Based Publish/Subscribe DHT Network". In: *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*. ICDCS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 23–.

[Bal+03] H. Balakrishnan et al. "Looking up data in P2P systems". In: *Commun. ACM* 46.2 (Feb. 2003), pp. 43–48.

[BC08] M. Becchi and P. Crowley. "Efficient regular expression evaluation: theory to practice". In: *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '08. San Jose, California: ACM, 2008, pp. 50–59.

[Bec07] Becchi, Michela and Crowley, Patrick. "An improved algorithm to accelerate regular expression evaluation". In: *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ANCS '07. Orlando, Florida, USA: ACM, 2007, pp. 145–154.

[Bro06] Brodie, Benjamin C. and Taylor, David E. and Cytron, Ron K. "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching". In: *SIGARCH Comput. Archit. News* 34.2 (May 2006), pp. 191–202.

[BS08] A. S. Bernard Wong and E. G. Sirer. *Approximate Matching for Peer-to-Peer Overlays with Cubit*. 2008.

[Cho56] N. Chomsky. "Three models for the description of language". In: *Information Theory, IRE Transactions on* 2.3 (Sept. 1956), pp. 113–124.

[Coo12] Cooperative Association for Internet Data Analysis. *CAIDA Routeviews Prefix to AS mappings Dataset @online*. Sept. 2012. URL: http://www.caida.org/data/routing/routeviews-prefix2as.xml.

[EG11#1] N. S. Evans and C. Grothoff. "Beyond Simulation: Large-Scale Distributed Emulation of P2P Protocols". In: *4th Workshop on Cyber Security Experimentation and Test (CSET 2011)*. USENIX Association. San Francisco, California: USENIX Association, 2011.

[EG11#2] N. S. Evans and C. Grothoff. "R5N : Randomized Recursive Routing for Restricted-Route Networks". In: *5th International Conference on Network and System Security (NSS 2011)*. IEEE. Milan, Italy: IEEE, Sept. 2011.

[Gar+04] L. Garcés-Erice et al. "Data Indexing in Peer-to-Peer DHT Networks". In: *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. ICDCS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 200–208.

[GL02] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59.

[GNU12] GNU Developer Team. *GNU C Regular Expression Library @online*. Oct. 2012. URL: http://www.gnu.org/software/libc/manual/html_node/Regular-Expressions.html.

[Gra12] Graphviz. *Graphviz - Graph Visualization Software @online*. Oct. 2012. URL: http://www.graphviz.org.

[Gro12] Grothoff, Christian. *GNU's Framework for Secure Peer-to-Peer Networking @online*. Sept. 2012. URL: https://gnunet.org.

[HMU07] J. E. Hopcroft, R. Motwani and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 2007.

[Kar+06] M. Karnstedt et al. "Similarity Queries on Structured Data in Structured Overlays". In: *Proceedings of the 22nd International Conference on Data Engineering Workshops*. ICDEW '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 32–.

[Kum06] Kumar, Sailesh and Dharmapurikar, Sarang and Yu, Fang and Crowley, Patrick and Turner, Jonathan. "Algorithms to accelerate multiple regular expressions matching for deep packet inspection". In: *SIGCOMM Comput. Commun. Rev.* 36.4 (Aug. 2006), pp. 339–350.

[Lee+07] J. Lee et al. "CISS: An efficient object clustering framework for DHT-based peer-to-peer applications". In: *Comput. Netw.* 51.4 (Mar. 2007), pp. 1072–1094.

[Neu05] C. Neumann. "Converting Deterministic Finite Automata to Regular Expressions". In: (2005).

[Rat+01] S. Ratnasamy et al. "A Scalable Content-Addressable Network". In: *IN PROC. ACM SIGCOMM 2001*. 2001, pp. 161–172.

[Sak09] J. Sakarovitch. *Elements of Automata Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2009.

[She+09] X. Shen et al. *Handbook of Peer-to-Peer Networking*. Lecture Notes on Coastal and Estuarine Studies. Springer, 2009.

[Sin06] Y. Singh. *Mathematical Foundation Of Computer Science*. New Age International (P) Limited, 2006, pp. 267–271.

[Sip06] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006, pp. 31–77.

[Sri12] Srisuresh, P. et al. *RFC-2694: DNS extensions to Network Address Translators (DNS_ALG) @online*. Sept. 2012. URL: http://tools.ietf.org/html/rfc2694.

[TUM12] TUM Lehrstuhl für Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur. *INFINIBAND at LRR-TUM @online*. Oct. 2012. URL: http://www.lrr.in.tum.de/Par/arch/infiniband/.

[Uni12] University of Oregon. *University of Oregon Route Views Project @online*. Sept. 2012. URL: http://www.routeviews.org/.

[War12] Warren, Paul. *RFC-822 compliant regular expression @online*. Sept. 2012. URL: http://ex-parrot.com/~pdw/Mail-RFC822-Address.html.

APPENDIX

A PROFILING

A.1 DATABASE INFORMATION FOR SIMULATED EVALUATION

A.1.1 *Table Schema*

```
CREATE TABLE `test_100` (
  `id` int NOT NULL AUTO_INCREMENT,
  `key` BINARY(64) NOT NULL,
  `label` varchar(32) DEFAULT NULL,
  `to_key` BINARY(64) DEFAULT NULL,
  `accepting` tinyint(1) DEFAULT '0',
  PRIMARY KEY (`id`),
  INDEX idx_key (`key`(64)),
  INDEX idx_key_id (`key`(64),id),
  INDEX idx_key_lbl_to_key (
    `key`(64), `label`(32), `to_key`(64)),
  INDEX idx_key_acc (`key`(64), `accepting`),
  UNIQUE KEY `unique` (
    `key`(64), `label`, `to_key`(64), `accepting`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

A.1.2 *Prepared Insert Statement*

```
INSERT IGNORE INTO NFA
(`key`, `label`, `to_key`, `accepting`)
VALUES (?, ?, ?, ?);
```

A.2 PROFILING TOOLS REFERENCE

In this section the usage of the profiling tools is described. Important options are highlighted.

A.2.1 *gnunet-regex-simulation-profiler*

```
gnunet-regex-simulationprofiler [OPTIONS] policy-dir
Profiler for regex library
Arguments mandatory for long options are also mandatory for short options.
  -c, --config=FILENAME      use configuration file FILENAME
  -h, --help                 print this help
  -L, --log=LOGLEVEL         configure logging to use LOGLEVEL
  -l, --logfile=LOGFILE      configure logging to write logs to LOGFILE
  -p, --max-path-compression=LENGTH
                             maximum path compression length
  -s, --stride-len=LENGTH    length of additional multi strides in the DFA
  -t, --table=TABLENAME      name of the table to write DFAs
  -v, --version              print the version number
Report bugs to gnunet-developers@gnu.org.
GNUnet home page: http://www.gnu.org/software/gnunet/
General help using GNU software: http://www.gnu.org/gethelp/
```

The configuration file, specified with the `-c` option should contain the information needed to connect to a MySQL database with the schema shown above. The following is an example for such a configuration file.

```
[regex-mysql]
DATABASE = dbname
USER = dbuser
PASSWORD = dbpass
HOST = localhost
PORT = 3306
```

A.2.2 *gnunet-regex-profiler*

```
gnunet-regex-profiler [OPTIONS] hosts-file policy-dir search-strings-file
Profiler for regex/mesh
Arguments mandatory for long options are also mandatory for short options.
  -a, --num-search-strings=COUNT
                            number of search strings to read from search
                              strings file
  -c, --config=FILENAME     use configuration file FILENAME
  -d, --details=FILENAME    name of the file for writing statistics
  -h, --help                print this help
  -L, --log=LOGLEVEL        configure logging to use LOGLEVEL
  -l, --logfile=LOGFILE     configure logging to write logs to LOGFILE
  -n, --num-links=COUNT     create COUNT number of random links
                            between peers
  -p, --max-path-compression=MAX_PATH_COMPRESSION
                            maximum path compression length
  -s, --search-delay=DELAY  wait DELAY before starting string search
  -t, --matching-timeout=TIMEOUT
                            wait TIMEOUT before considering a string
                            match as failed
  -v, --version             print the version number
Report bugs to gnunet-developers@gnu.org.
GNUnet home page: http://www.gnu.org/software/gnunet/
General help using GNU software: http://www.gnu.org/gethelp/
```

B

REGEX LIBRARY

B.1 EXAMPLE REGEX DFA

```
GNVPN-0001-PAD(0011101010010010(0|1)*|0011101010010010000(0|1)*
|0011101010010010001(0|1)*|0011101010010010010(0|1)*|0011101010010010011(0|1)*
|0011101010010010100(0|1)*|0011101010010010101(0|1)*|0011101010010010110(0|1)*
|0011101010010010111(0|1)*|0011101010110110(0|1)*|0011101010110110000000(0|1)*
|0011101010110110000001(0|1)*|0011101010110110000010(0|1)*|0011101010110110000011(0|1)*
|0011101010110110000100(0|1)*|0011101010110110000101(0|1)*|0011101010110110000110(0|1)*
|0011101010110110000111(0|1)*|0011101010110110001000(0|1)*|0011101010110110001001(0|1)*
|0011101010110110001010(0|1)*|0011101010110110001011(0|1)*|0011101010110110001100(0|1)*
|0011101010110110001101(0|1)*|0011101010110110001110(0|1)*|0011101010110110001111(0|1)*
|0011101010110110010000(0|1)*|0011101010110110010001(0|1)*|0011101010110110010010(0|1)*
|0011101010110110010011(0|1)*|0011101010110110010100(0|1)*|0011101010110110010101(0|1)*
|0011101010110110010110(0|1)*|0011101010110110010111(0|1)*|0011101010110110011000(0|1)*
|0011101010110110011001(0|1)*|0011101010110110011010(0|1)*|0011101010110110011011(0|1)*
|0011101010110110011100(0|1)*|0011101010110110011101(0|1)*|0011101010110110011110(0|1)*
|0011101010110111111(0|1)*|0011101110111101(0|1)*|00111011101111010000000(0|1)*
|0011101111011110100001(0|1)*|0011101111011110100010(0|1)*|0011101111011110100011(0|1)*
|0011101111011110100100(0|1)*|0011101111011110100101(0|1)*|0011101111011110100110(0|1)*
|0011101111011110100111(0|1)*|0011101111011110101000(0|1)*|0011101111011110101001(0|1)*
|0011101111011110101010(0|1)*|0011101111011110101011(0|1)*|0011101111011110101100(0|1)*
|0011101111011110101101(0|1)*|0011101111011110101110(0|1)*|0011101111011110101111(0|1)*
|0011101111011110110000(0|1)*|0011101111011110110001(0|1)*|0011101111011110110010(0|1)*
|0011101111011110110011(0|1)*|0011101111011110110100(0|1)*|0011101111011110110101(0|1)*
|0011101111011110110110(0|1)*|0011101111011110110111(0|1)*|0011101111011110111000(0|1)*
|0011101111011110111001(0|1)*|0011101111011110111010(0|1)*|0011101111011110111011(0|1)*
|0011101111011110111100(0|1)*|0011101111011110111101(0|1)*|0011101111011110111110(0|1)*
|0111010001010110(0|1)*|0111010001010110000000(0|1)*|0111010001010110000001(0|1)*
|0111010001010110000010(0|1)*|0111010001010110000011(0|1)*|0111010001010110000100(0|1)*
|0111010001010110000101(0|1)*|0111010001010110000110(0|1)*|0111010001010110000111(0|1)*
|0111010001010110001000(0|1)*|0111010001010110001001(0|1)*|0111010001010110001010(0|1)*
|0111010001010110001011(0|1)*|0111010001010110001100(0|1)*|0111010001010110001101(0|1)*
|0111010001010110001110(0|1)*|0111010001010110001111(0|1)*|0111010001010110010000(0|1)*
|0111010001010110010001(0|1)*|0111010001010110010010(0|1)*|0111010001010110010011(0|1)*
|0111010001010110010100(0|1)*|0111010001010110010101(0|1)*|0111010001010110010110(0|1)*
|0111010001010110010111(0|1)*|0111010001010110011000(0|1)*|0111010001010110011001(0|1)*
|0111010001010110011010(0|1)*|0111010001010110011011(0|1)*|0111010001010110011100(0|1)*
|0111010001010110011101(0|1)*|0111010001010110011110(0|1)*|0111010001010110011111(0|1)*
|0111010001010111(0|1)*|0111010001010111000000(0|1)*|0111010001010111000001(0|1)*
|0111010001010111000010(0|1)*|0111010001010111000011(0|1)*|0111010001010111000100(0|1)*
|0111010001010111000101(0|1)*|0111010001010111000110(0|1)*|0111010001010111000111(0|1)*
|0111010001010111001000(0|1)*|0111010001010111001001(0|1)*|0111010001010111001010(0|1)*
|0111010001010111001011(0|1)*|0111010001010111001100(0|1)*|0111010001010111001101(0|1)*
|0111010001010111001110(0|1)*|0111010001010111001111(0|1)*|0111010001010111010000(0|1)*
|0111010001010111010001(0|1)*|0111010001010111010010(0|1)*|0111010001010111010011(0|1)*
|0111010001010111010100(0|1)*|0111010001010111010101(0|1)*|0111010001010111010110(0|1)*
|0111010001010111010111(0|1)*|0111010001010111011000(0|1)*|0111010001010111011001(0|1)*
|0111010001010111011010(0|1)*|0111010001010111011011(0|1)*|0111010001010111011100(0|1)*
|0111010001010111011101(0|1)*|0111010001010111011110(0|1)*|0111010001010111011111(0|1)*
|0111010001010111(0|1)*|0111010001010111100000(0|1)*|0111010001010111100001(0|1)*
|0111010001010111100010(0|1)*|0111010001010111100011(0|1)*|0111010001010111100100(0|1)*
|0111010001010111100101(0|1)*|0111010001010111100110(0|1)*|0111010001010111100111(0|1)*
|0111010001010111101000(0|1)*|0111010001010111101001(0|1)*|0111010001010111101010(0|1)*
|0111010001010111101011(0|1)*|0111010001010111101100(0|1)*|0111010001010111101101(0|1)*
|0111010001010111101110(0|1)*|0111010001010111101111(0|1)*|0111010001010111110000(0|1)*
|0111010001010111110001(0|1)*|0111010001010111110010(0|1)*|0111010001010111110011(0|1)*
|0111010001010111110100(0|1)*|0111010001010111110101(0|1)*|0111010001010111110110(0|1)*
|0111010001010111110111(0|1)*|0111010001010111111000(0|1)*|0111010001010111111001(0|1)*
|0111010001010111111010(0|1)*|0111010001010111111011(0|1)*|0111010001010111111100(0|1)*
|0111010001010111111101(0|1)*|0111010001010111111110(0|1)*|0111010001010111111111(0|1)*
|0111010001011000(0|1)*|0111010001011000000000(0|1)*|0111010001011000000001(0|1)*
|0111010001011000000010(0|1)*|0111010001011000000011(0|1)*|0111010001011000000100(0|1)*
|0111010001011000000101(0|1)*|0111010001011000000110(0|1)*|0111010001011000000111(0|1)*
|0111010001011000001000(0|1)*|0111010001011000001001(0|1)*|0111010001011000001010(0|1)*
|0111010001011000001011(0|1)*|0111010001011000001100(0|1)*|0111010001011000001101(0|1)*
|0111010001011000001110(0|1)*|0111010001011000001111(0|1)*|0111010001011000010000(0|1)*
|0111010001011000010001(0|1)*|0111010001011000010010(0|1)*|0111010001011000010011(0|1)*
|0111010001011000010100(0|1)*|0111010001011000010101(0|1)*|0111010001011000010110(0|1)*
|0111010001011000010111(0|1)*|0111010001011000011000(0|1)*|0111010001011000011001(0|1)*
|0111010001011000011010(0|1)*|0111010001011000011011(0|1)*|0111010001011000011100(0|1)*
|0111010001011000011101(0|1)*|0111010001011000011110(0|1)*|0111010001011000011111(0|1)*
```

```
|011101000101100100000(0|1)*|011101000101100100001(0|1)*
|011101000101100100010(0|1)*|011101000101100100011(0|1)*|011101000101100100100(0|1)*
|011101000101100100101(0|1)*|011101000101100100110(0|1)*|011101000101100100111(0|1)*
|011101000101100101000(0|1)*|011101000101100101001(0|1)*|011101000101100101010(0|1)*
|011101000101100101011(0|1)*|011101000101100101100(0|1)*|011101000101100101101(0|1)*
|011101000101100101110(0|1)*|011101000101100101111(0|1)*|011101000101100101111000(0|1)*
|1100101010011100(0|1)*|1100101010011100000000(0|1)*|1100101010011100000000001(0|1)*
|1100101010011100000000010(0|1)*|1100101010011100000000110(0|1)*|1100101010011100000001(0|1)*
|1100101010011100000001000(0|1)*|1100101010011100000001001(0|1)*|1100101010011100000001010(0|1)*
|1100101010011100000001011(0|1)*|1100101010011100000001101(0|1)*|1100101010011100000001110(0|1)*
|1100101010011100000010(0|1)*|1100101010011100000011(0|1)*|1100101010011100000100(0|1)*
|1100101010011100000101(0|1)*|1100101010011100000110(0|1)*|1100101010011100000111(0|1)*
|1100101010011100001000(0|1)*|1100101010011100001001(0|1)*|1100101010011100001010(0|1)*
|1100101010011100001011(0|1)*|1100101010011100001100(0|1)*|1100101010011100001101(0|1)*
|1100101010011100001110(0|1)*|1100101010011100001111(0|1)*|1100101010011100010000(0|1)*
|1100101010011100010001(0|1)*|1100101010011100010010(0|1)*|1100101010011100010011(0|1)*
|1100101010011100010100(0|1)*|1100101010011100010101(0|1)*|1100101010011100010110(0|1)*
|1100101010011100010111(0|1)*|1100101010011100011000(0|1)*|1100101010011100011001(0|1)*
|1100101010011100011010(0|1)*|1100101010011100011011(0|1)*|1100101010011100011100(0|1)*
|1100101010011100011101(0|1)*|1100101010011100011110(0|1)*|1100101010011100011111(0|1)*
|1101101010111010(0|1)*|1101101010111010000000(0|1)*|1101101010111010000000001(0|1)*
|1101101010111010000001000(0|1)*|1101101010111010000001001(0|1)*|1101101010111010000001010(0|1)*
|1101101010111010000001011(0|1)*|1101101010111010000001100(0|1)*|1101101010111010000001110(0|1)*
|1101101010111010000001111(0|1)*|1101101010111010000010(0|1)*|1101101010111010000010000(0|1)*
|1101101010111010000010001(0|1)*|1101101010111010000010010(0|1)*|1101101010111010000010011(0|1)*
|1101101010111010000010100(0|1)*|1101101010111010000010101(0|1)*|1101101010111010000010110(0|1)*
|1101101010111010000010111(0|1)*|1101101010111010000011000(0|1)*|1101101010111010000011001(0|1)*
|1101101010111010000011010(0|1)*|1101101010111010000011011(0|1)*|1101101010111010000011100(0|1)*
|1101101010111010000011101(0|1)*|1101101010111010000011110(0|1)*|1101101010111010000011111(0|1)*
|1101101010111010000100(0|1)*|1101101010111010000101(0|1)*|1101101010111010000110(0|1)*
|1101101010111010000111(0|1)*|1101101010111010001000(0|1)*|1101101010111010001001(0|1)*
|1101101010111010001010(0|1)*|1101101010111010001011(0|1)*|1101101010111010001100(0|1)*
|1101101010111010001101(0|1)*|1101101010111010001110(0|1)*|1101101010111010001111(0|1)*
|1101101010111010010000(0|1)*|1101101010111010010001(0|1)*|1101101010111010010010(0|1)*
|1101101010111010010011(0|1)*|1101101010111010010100(0|1)*|1101101010111010010101(0|1)*
|1101101010111010010110(0|1)*|1101101010111010010111(0|1)*|1101101010111010011000(0|1)*
|1101101010111010011001(0|1)*|1101101010111010011010(0|1)*|1101101010111010011011(0|1)*
|1101101010111010011100(0|1)*|1101101010111010011101(0|1)*|1101101010111010011110(0|1)*
|1101101010111010011111(0|1)*|1101101010111010100(0|1)*|1101101010111010101(0|1)*
|1101101010111010110(0|1)*|1101101010111010111(0|1)*|1101101010111011000(0|1)*
|1101101010111011001(0|1)*|1101101010111011010(0|1)*|1101101010111011011(0|1)*
|1101101010111011100(0|1)*|1101101010111011101(0|1)*|1101101010111011110(0|1)*
|1101101010111011111(0|1)*|1101101011010100(0|1)*|1101101011010100000(0|1)*
|1101101011010100001(0|1)*|1101101011010100010(0|1)*|1101101011010100011(0|1)*
|1101101011010100100(0|1)*|1101101011010100101(0|1)*|1101101011010100110(0|1)*
|1101101011010100111(0|1)*|1101101011010101000(0|1)*|1101101011010101001(0|1)*
|1101101011010101010(0|1)*|1101101011010101011(0|1)*|1101101011010101100(0|1)*
|1101101011010101101(0|1)*|1101101011010101110(0|1)*|1101101011010101111(0|1)*
|1101101011010110000(0|1)*|1101101011010110001(0|1)*|1101101011010110010(0|1)*
|1101101011010110011(0|1)*|1101101011010110100(0|1)*|1101101011010110101(0|1)*
|1101101011010110110(0|1)*|1101101011010110111(0|1)*|1101101011010111000(0|1)*
|1101101011010111001(0|1)*|1101101011010111010(0|1)*|1101101011010111011(0|1)*
|1101101011010111100(0|1)*|1101101011010111101(0|1)*|1101101011010111110(0|1)*
|1101101011010111111(0|1)*|1101110101000000(0|1)*|1101110101000000001(0|1)*
|1101110101000001000(0|1)*|1101110101000001001(0|1)*|1101110101000001010(0|1)*
|1101110101000001011(0|1)*|1101110101000001100(0|1)*|1101110101000001101(0|1)*
|1101110101000001110(0|1)*|1101110101000001111(0|1)*|1101110101000010000(0|1)*
|1101110101000010001(0|1)*|1101110101000010010(0|1)*|1101110101000010011(0|1)*
|1101110101000010100(0|1)*|1101110101000010101(0|1)*|1101110101000010110(0|1)*
|1101110101000010111(0|1)*|1101110101000011000(0|1)*|1101110101000011001(0|1)*
|1101110101000011010(0|1)*|1101110101000011011(0|1)*|1101110101000011100(0|1)*
|1101110101000011101(0|1)*|1101110101000011110(0|1)*|1101110101000011111(0|1)*
|1101110101000100(0|1)*|1101110101000101(0|1)*|1101110101000110(0|1)*
|1101110101000111(0|1)*|1101110101001000(0|1)*|1101110101001001(0|1)*
|1101110101001010(0|1)*|1101110101001011(0|1)*|1101110101001100(0|1)*
|1101110101001101(0|1)*|1101110101001110(0|1)*|1101110101001111(0|1)*
|1101110101010000(0|1)*|1101110101010001(0|1)*|1101110101010010(0|1)*
|1101110101010011(0|1)*|1101110101010100(0|1)*|1101110101010101(0|1)*
|1101110101010110(0|1)*|1101110101010111(0|1)*|1101110101011000(0|1)*
|1101110101011001(0|1)*|1101110101011010(0|1)*|1101110101011011(0|1)*
|1101110101011100(0|1)*|1101110101011101(0|1)*|1101110101011110(0|1)*
|1101110101011111(0|1)*)"
```

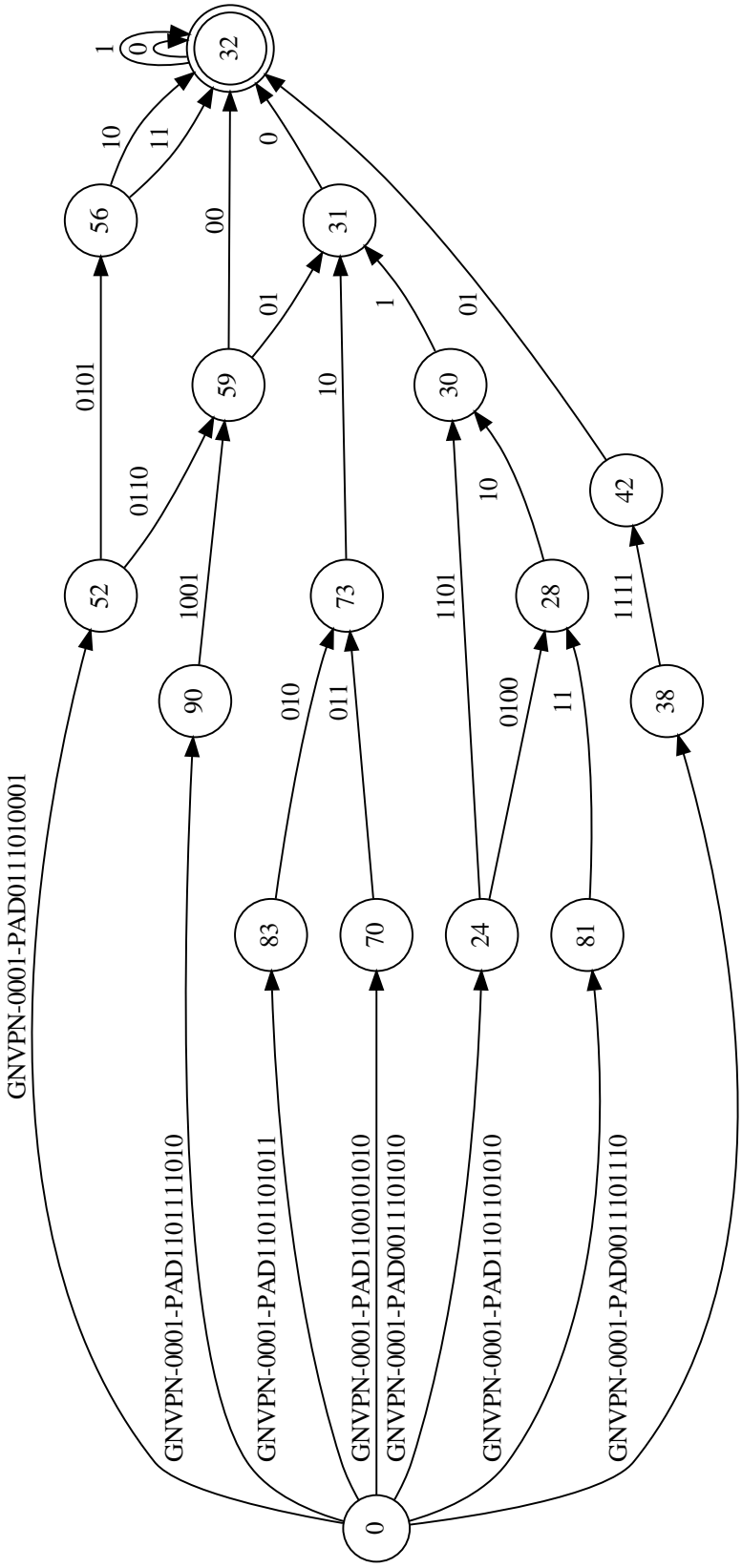


Figure B.1: DFA for the above regular expression describing an AS, generated using the GNUnet Regex Library.

C

WEB RESOURCES

C.1 RFC-822 COMPLIANT REGULAR EXPRESSION

A regular expression that is compliant to the RFC-822 standard according to [War12].

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
\t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n) ?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(
?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;
:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\"
.\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\
]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:,@(?:(?:\r\n)?[ \t])*)(?:[^()<>@,;:\\".\
[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]
|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \0
00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\
.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,
;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?
:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[
^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]
```

```
]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*(
?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(
?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t
])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?
:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|
\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\
]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)
?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["
()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)
?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>
@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[
 \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,
;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[ \t])*)?
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:
\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])
*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\
.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(
?:\r\n)?[ \t])*))*)?;\s*)
```