Efficient Tree-Based Content-Based Routing Schemes

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Koorosh Khazaei

under the supervision of
Prof. Antonio Carzaniga

February 2018

Dissertation Committee

Prof. Fernando Pedone	Università della Svizzera Italiana, Switzerland
Prof. Patrick Eugster	Università della Svizzera Italiana, Switzerland
Prof. Fabian Kuhn	University of Freiburg, Germany
Prof. Pascal Felber	Université de Neuchâtel, Switzerland

Dissertation accepted on 5 February 2018

Research Advisor	PhD Program Director
Prof. Antonio Carzaniga	**Prof. Walter Binder**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Koorosh Khazaei
Lugano, 5 February 2018

To my lovely wife, Paniz Afroughi

I do not pretend to start with precise questions. I do not think you can start with anything precise. You have to achieve such precision as you can, as you go along.

Bertrand Russell

Abstract

This thesis is about routing and forwarding for inherently multicast communication such as the communication typical of information-centric networks.

The notion of Information-Centric Networking (ICN) is an evolution of the Internet from the current host-centric architecture to a new architecture in which communication is based on "named information". The ambitious goal of ICN is to effectively support the exchange and use of information in an ever more connected world, with billions of devices, many of which are mobile, producing and consuming large amounts of data. ICN is intended to support scalable content distribution, mobility, and security, for such applications as video on demand and networks of sensors or the so-called Internet of Things.

Many ICN architectures have emerged in the past decade, and the ICN community has made significant progress in terms of infrastructure, test-bed deployments, and application case studies. And yet, despite the impressive research effort, the fundamental problems of routing and forwarding remain open. In particular, none of the proposed architectures has developed truly scalable name-based routing schemes and efficient name-based forwarding algorithms.

This is not surprising, since the problem of routing based on names, in its most general formulation, is known to be fundamentally difficult. In general, one would want to support application-defined names (as opposed to network-defined addresses) with a compact routing scheme (small routing tables) that uses optimal paths and minimizes congestion, and that admits to a fast forwarding algorithm. Furthermore, one would want to construct this routing scheme with a decentralized and incremental protocol for administrative autonomy and efficient dynamic updates. However, there are clear theoretical limits that simply make it impossible to achieve all these goals.

In this thesis we explore the design space of routing and forwarding in an information-centric network. Our purpose is to develop routing schemes and forwarding algorithms that combine many desirable properties. We consider two forms of addressing, one tied to network locations, and one based on more expressive content descriptors. We then consider trees as basic routing struc-

tures, and with those we develop routing schemes that are intended to minimize path lengths and congestion, separately or together. For one of these schemes based on expressive content descriptors, we also develop a fast forwarding algorithm specialized for massively parallel architectures such as GPUs.

In summary, this thesis presents two efficient and scalable routing algorithms for two different types of networks, plus one scalable forwarding algorithm. We summarize each individual contribution below:

- *Low-congestion geographic routing for wireless networks.* We develop a low-congestion, multicast routing scheme designed specifically for wireless networks. The scheme supports *geographical* multicast routing, meaning routing to a set of nodes addressed by their physical position. The scheme builds a geometric minimum spanning tree connecting the source to all the destinations. Then, for each edge in this tree, the scheme routes a message through a random intermediate node, chosen independently of the set of multicast requests. The intermediate node is chosen in the vicinity of the corresponding edge such that congestion is reduced without stretching routes by more than a constant factor.

- *Multi-tree scheme for content-based routing in ICN.* We develop a tree-based routing scheme designed for large-scale wired networks such as the Internet. The scheme supports two forms of addresses: application-defined content *descriptors,* and network-defined *locators.* We first show that the scheme is effective in terms of stretch and congestion on the current AS-level Internet graph even with only a few spanning trees. Then we show that our content descriptors, which consist of sets of tags and that are more expressive than the name prefixes used in mainstream ICN, aggregate well in practice under our scheme. We also explain in detail how to use descriptors and locators, together with unique content identifiers, to support the efficient transmission and sharing of information through scalable and loop-free routes.

- *Tag-based forwarding (partial matching) algorithm on GPUs.* To accompany our ICN routing scheme, we develop a fast forwarding algorithm that matches incoming packets against forwarding tables with tens of millions of entries. To achieve high performance, we develop a practical solution for the *partial matching* problem that lies at the heart of this forwarding scheme. This solution amounts to a massively parallel algorithm specifically designed for a hybrid CPU/GPU architecture.

Acknowledgments

I would like to express my deepest appreciation and gratitude to my advisor and friend Antonio Carzaniga for all the support and friendship you have given me over the years. This dissertation would not have been possible without your guidance and patience. I would not be who I am or where I am today if I had not had the privilege to know and work with you.

Contents

Figures

Tables

Algorithms

Chapter 1

Introduction

The current Internet implements a communication model based on "host" addresses that are essentially location-dependent and in any case network-defined. With these addresses, the network supports almost exclusively unicast information flows. This communication model might have satisfied the communication needs of applications from the early stages of the Internet. However, today's end-user applications use the network not to access or connect to any specific host, but rather to exchange information or perhaps to connect two or more users. Similarly, other applications, such as data-center applications, are also based primarily on content-driven data flows or on symbolic services, not on pure host-based communication.

Modern applications also require multicast flows. An example of an application where content has to be delivered to multiple receivers is video conferencing or live video streaming. As it turns out, the vast majority of the current Internet traffic—a "high 90% level of traffic"—consists of data disseminated from a source to a number of users.[1] In other words, most Internet traffic is essentially multicast. Such multicast flows can be supported through unicast flows, although the redundancy of such flows leads to inefficiencies and possibly to congestion. A good network-level multicast primitive may overcome this problem, but current multicast solutions such as IP multicast are available almost exclusively to network operators but not to users and content providers.

In short, there is a gap between the needs of applications and the basic host-based unicast network service available at the network level. Many application-level communication systems (middleware) are already there to fill this gap. One such system that is ubiquitous and very successful—and also essential—is the domain name system (DNS). Still, DNS provides access to a content space

[1] Van Jacobson. *A New Way to look at Networking.* Google Tech Talk, August 30, 2006.

(domain names) that is very static and arguably very far from applications, in the sense that applications have no control over it. So, we start from the presupposition that there is still a significant gap, and that new applications require or at least would greatly benefit from richer modes of communication.

One such novel mode of communication is what is embodied in Information-Centric Networking (ICN). The main purpose of an information-centric network is to transmit information rather than to connect to specific hosts. Thus a consumer would request the desired information, and the network would deliver it, regardless of the host where the information originated or where it might be stored. Information centric networking is also inherently multicast, in the sense that information would naturally flow to all interested consumers.

This thesis addresses some fundamental problems in information-centric and multicast communication.

In Chapter 4 we focus our effort specifically on an optimal solution for the congestion problem in multicast communication. Beyond that, we continue to pay special attention to multicast communication and congestion throughout the rest of this thesis. In fact, we consider multicast communication as a fundamental requirement for our proposed ICN routing schemes.

Congestion and multicast are not the only concerns in designing new communication schemes. Our general goal is to provide an communication service that is close to the needs of applications. As we argued so far, many applications are most fundamentally interested in *information*, regardless of the network location where that information may reside, and regardless of the network-defined address to which that information may be sent. In other words, such applications would benefit from a content-based addressing model whereby information is delivered to any and only those hosts that are interested in that content. And once again, this is one of the primary goals of information-centric networking (ICN).

Therefore the design of ICN includes provisions for scalability in content distribution, as well as features that the current Internet lacks, and especially features suitable for modern Internet applications. An ideal ICN design would natively support mobility, multi-point access, and multi-homing, as well as anycast, multicast, and broadcast. Data would become independent from location, application, storage, and means of transportation, enabling in-network caching and replication. ICN has purported benefits also for security, specifically for authentication. In particular, since information could be addressed by unique identifiers visible and understood at the network level, information could also be equipped with appropriate signatures which would also allow the network itself to verify authenticity on the behalf of users applications. Finally, these

ICN communications services should be scalable to support billions of devices that produce and consume large amounts of digital information.

Such an ICN design is quite ambitious and, perhaps not surprisingly, its realization has been difficult. Many ICN architectures have emerged in the past decade, but they mostly focused on a small subset of the goals envisioned for ICN. In particular, none of the proposed architectures has developed scalable name-based, let alone true *content*-based routing schemes and corresponding efficient forwarding algorithms, which must be the pillars of any ICN scalable communication scheme. These are the problems we attempt to solve with this thesis.

1.1 The Importance of Scalability

The current Internet is very different from its early days. In terms of size, it is estimated that currently 15 billion devices are connected to the Internet. Cisco estimates that 50 billion devices and objects will be connected to the Internet by 2020[2], the Internet of Things World Forum (IoTWF) estimated this number to reach 82 billion in 2025 [3], and Intel estimates 200 billion devices by 2020.[4]

Regardless of whose estimate will be closer to reality, the so-called Internet of Things (IoT) is expected to contribute a lot to these numbers. Moreover, so many connected devices, such as smart phones, can *produce* as well as consume huge amounts of data. This creates a new demand for the network. Namely, the network should provide easy ways for consumers to get the content they are interested in also from this new type of producers. However, if content is addressed directly at the network level, allowing every node to be a content producer would have a significant impact on the size of the Routing Information Base (RIB). This is perhaps the most crucial aspect of scalability for an information centric network. In fact, many proposed ICN architectures do not allow this to happen, mainly due to the fact that their routing algorithms are either based on link-state or distance-vector routing and can not cope with the magnitude and distribution of user-produced content.

Scalability, specifically for highly distributed application-defined content descriptors, is an explicit and fundamental goal for this thesis.

[2] http://www.cisco.com/web/solutions/trends/iot/overview.html

[3] IDC Worldwide Internet of Things Installed Base by Connectivity Forecast, March 2017. Link:https://www.idc.com/getdoc.jsp?containerId=US42331917

[4] http://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html

1.2 The Importance of the Naming Scheme

Most of the current proposed architectures for ICN, including the mainstream CCN and NDN projects, use a hierarchical naming scheme to address content. As we show in more detail later in Section 2.2.1, this way of naming information has some important limitations. First of all, hierarchical names impose a fixed hierarchical partitioning of the content space that might be too restrictive for many applications. Also, hierarchical names may or may not aggregate well. Names aggregate by common prefix, which might not lead to an effective aggregation with data produced by several applications dispersed throughout the network, especially when names are unrelated to the network topology. This is why many ICN implementations disallow true network-independent naming and instead impose network-specific prefixes similar to DNS names, thereby limiting even more the expressiveness of the naming scheme from the viewpoint of applications.

We consider the expressiveness of the naming scheme to be of fundamental importance. However, we also see the difficulty of scaling up routing for very expressive names. Therefore, in this thesis we propose a dual naming scheme that combines expressive content *descriptors* consisting of sets of tags, which are application defined and therefore may or may not aggregate well, with opaque *locators*, which are network defined and therefore can be very efficient for routing. We then develop a routing scheme and a forwarding algorithm for these two schemes.

1.3 Contributions of This Thesis

There are a number of important objectives that every good routing scheme tries to achieve. The first objective is to obtain routing paths that "stretch" as little as possible beyond the optimal distance. This is important since the stretch of routing paths has a direct impact on the efficiency of the communication in the network. The second objective is to incur low congestion in the network. A third important goal for a good routing scheme is to reduce the amount of memory required at each node of the network to maintain the routing paths. In summary one would want a routing scheme that has low stretch, low congestion, and low memory utilization.

Unfortunately, for most network models and routing algorithms, there is a trade-off between every two of these objectives. These trade-offs for routing algorithms have been studied extensively for the unicast communication

model. For example the trade-offs between memory and stretch are well studied under the rubric of "compact routing". There are also routing algorithms for congestion optimization (see Section 1.4).

Researchers usually study one of these trade-offs in isolation in order to make the study more tractable. Therefore, the research literature is very limited regarding the development of routing schemes that consider all three trade-offs (stretch, congestion, memory)). This is especially the case for the ICN communication model, which is the main focus of this thesis.

Moreover, the focus of this research is to design and develop ICN routing schemes that are not only efficient, again in terms of memory stretch and congestion, but also practical. In order to be practical, a routing scheme must be *distributed*, meaning amenable to a decentralized construction; it must be *online*, in the sense of supporting application demands that are not completely known in advance; it must be *dynamic*, in the sense of supporting evolving networks and/or naming schemes; and finally it should be accompanied by a fast forwarding algorithm (see Section 1.4). We summarize the desired properties of an ideal routing scheme in the first row of Table 1.1. On the basis of these desirable properties, we then characterize the contribution of this thesis, comparing the schemes and algorithms developed here with the ideal routing scheme.

The trade-offs between stretch, congestion, and memory usage are fundamental. In other words, there are theoretical limits to achieving optimality in all of these objectives in the most general network model. Hence our approach is to study specific networks and communication models with special properties. This helps us to exploit those properties and to design routing schemes with minimal stretch, low congestion and low memory usage.

A fundamental choice in our approach is to work on tree-based routing schemes. A tree is an ideal structure to reach multiple destinations. It is loop-free by construction, and admits to an efficient forwarding of unicast or multicast packets. Routing on trees is particularly suitable for network models such as geometric networks, where routing trees can be built based on the location of nodes. Trees are also good in networks such as the Internet that are already structurally similar to a tree. Furthermore, for such networks it is also easy to maintain a hierarchy of trees spanning the network at different levels (e.g., inter-AS and intra-AS). Finally, since tree structures are very well studied, we can benefit from a trove of knowledge and existing algorithms.

This thesis makes three contributions in the study and design of information-centric networking.

routing scheme	network	service model	features
ideal scheme	any	unicast multicast ICN name-independent	compact low stretch low congestion fast forwarding on-line distributed dynamic
tree-based (Chapter 2)	internet inter-AS intra-AS	tag-based ICN name-independent	compact (partially) low stretch low congestion on-line dynamic (partially) hierarchical
geo-multicast (Chapter 4)	geographical unit-disc wireless	multicast name-dependent	compact low stretch low congestion (opt.) on-line dynamic oblivious

Table 1.1. Properties of the Routing Schemes Considered in This Thesis

1. *Multi-tree routing scheme for content-based routing in ICN.* We develop a practical approach to the routing problem in information-centric networking. At a very high-level, we consider a network model corresponding to an unweighted AS-level Internet topology, and focus on application-defined, tag-based addressing. Since the Internet topology, especially at the AS-level, is already structurally similar to a tree, we build a routing scheme using trees.

   Trees simplify many routing problems, especially for multicast routing. However, trees also have clear disadvantages for routing. First, paths might be stretched, meaning the distance between two nodes on the tree might be longer than their distance on the full graph. Second, traffic would flow only on the tree, thereby reducing the overall network throughput. It is well known that these sorts of problems can be alleviated by using multiple trees, each with their own forwarding state, but in principle that requires more memory. This question of memory

complexity is one of the fundamental issues we examine in this thesis.

We first show experimental results that indicate that multi-tree routing is a good candidate for information centric Networking on the Internet AS-level network [22]. Then we show that we can achieve different individual objectives, such as low memory requirement, low stretch, and low congestion by changing the way we construct these trees (see Section 2.3). Properties of this routing scheme are summarized in the second row of Table 1.1.

2. *Tag-based forwarding (partial matching) algorithm on GPUs.* Since a fast forwarding algorithm is a fundamental part of any good routing scheme, we put a lot of effort in designing one for the routing scheme we developed based on trees. We adopt a tag-based model for content descriptors (detailed in Section 1.4), which means that our forwarding algorithm amounts to a very fast solution for the *subset query* problem, which in turn is equivalent to the *partial matching* problem. Briefly, given a set $\mathcal{P}$ of N subsets of a universe U, $|U| = m$ and a query set $Q \subset U$, we have to find all $P \in \mathcal{P}$ such that $P \subseteq Q$. We give formal and precise definitions for this problem in Section 3.1.1.

   While this problem is well studied theoretically [73, 27] and practically (e.g. in the the database literature), existing solutions are not suitable to be used as a forwarding algorithm, especially for forwarding tables of the magnitude we expect in the ICN context. Solutions such as linear scan, signature-based methods, inverted-indexes, tries and other theoretical solutions such as [27], either consume too much memory that forces them to store the data structure on disk, or use too much computational power that makes them slow and not suitable for our problem domain. In fact, this problem is believed to be inherently difficult due to the "curse of dimensionality" [15] which means that there is no algorithm for this problem which achieves both "fast" query time and "small" space.

   We expect an ICN forwarding table to contain tens of millions of entries. Therefore we need to design a solution for this specific problem-size that does not consume too much memory and yet can achieve an acceptable performance when matching incoming packets against large forwarding tables. To achieve this goal, we develop a system that we call TagMatch that in essence scans the problem space linearly for a single query. To achieves high performance it processes many queries in a highly parallel fashion. TagMatch is composed of several components each optimized so

as to achieve high performance. TagMatch is designed to benefit from the power of both CPUs and general-purpose GPUs (GPGPU).

The "pre-process" stage on the CPU side performs an initial filtering of the incoming packets and assigns them to multiple queues while "subset match" component that runs on GPU processes each queue in parallel to find all the matches and reports back the result to the CPU for further processing. Chapter 3 details to specifics of this system and the evaluation of its performance.

3. *Low-congestion geographic routing for wireless networks.* While our general goal is to design information-centric routing schemes, in particular with content-based addressing and with all the desired features and properties as shown in the first row of Table 1.1, we started this research with a simpler routing scheme where we put restrictions on both the communication and network models. We present this contribution in the latter part of the thesis even though chronologically it was developed earlier.

   For the network model, in this case we consider unit-disc communication in geographical networks to model ad-hoc wireless networks. Because of the nice properties of this network model, there is no need to construct a fixed number of spanning trees to route multicast requests. Instead, we build multicast trees on-the-fly as a message moves toward its destinations.

   Based on this model, we propose and analyze an oblivious randomized multicast routing scheme that is compact (low memory) and has low stretch and low congestion [21]. Properties of this routing scheme are summarized in the third row of Table 1.1.

In the next section in this introductory chapter we describe the preliminary notions and terminology used throughout the rest of the thesis. After that, in Section 1.5, we outline the structure of this dissertation.

1.4 Preliminaries

In this section we review some basic notions regarding routing. We start with the brief definitions of many essential terms and concepts that are frequently used throughout this thesis. We then elaborate a bit on the communication model, the network model, and routing.

Network A network is a collection of processors interconnected by communication channels that allow them to communicate with each other. We model a network by a corresponding graph in which processors are represented by nodes and communication channels or links are represented by bidirectional edges between nodes. Processors host applications and also act as routers. Thus a router v can send and receive messages (or packets) to and from its $\delta(v)$ neighbors through $\delta(v)$ *interfaces* numbered $1 \ldots \delta(v)$. For uniformity, it is convenient to model the communications of applications running on a node v as messages going through a special interface number 0. Therefore, a message originating at node v_0 is implicitly received by v_0 through its interface 0, while one being delivered to an application on v_0 is forwarded through interface 0.

Path A message originating at node v_0 and delivered to node v_k is forwarded by a series of intermediate routers $v_1, v_2, \ldots, v_{k-1}$. The sequence of edges $(v_0, v_2), (v_1, v_2), \ldots, (v_{k-1}, v_k) \ldots$ define a *path* from node v_0 to v_k, and the number of edges in the sequence is called the *length* of the path.

Routing scheme A *routing scheme* is a distributed algorithm that installs forwarding state (or a forwarding information base, or FIB) in each router.

Forwarding scheme A corresponding *forwarding scheme* is a local decision procedure that executes at each router v whenever v receives a message and, given the message m and v's FIB, determines a set of output interfaces to which the router then forwards a copy of the input message. A forwarding scheme may also rewrite forwarded message.

Addressing scheme The addressing scheme is the way the originator of a message indicates one or more recipients for the message. This can be done either *explicitly* by specifying the recipients, or *implicitly* by describing the content of the message itself and then letting the network deliver a message to interested recipients. With an explicit addressing scheme, each node in the network is assigned a name, while with implicit addressing each node describes the kind of messages or information they intend to receive. Different addressing schemes enable the network to support different types of communication models. Routing and forwarding schemes are both based on the addressing scheme.

- *Name-dependent vs. name-independent addressing:* With an explicit addressing scheme, each node in the network must be given a name,

and each message contains the exact name of its destination node. Some routing schemes can work with whatever names, perhaps assigned by applications independently of the network topology. Such schemes are said to be *name-independent.* Other routing schemes require that names be assigned in a certain way, typically by the the routing scheme itself. These latter schemes are said to be *name-dependent* or *labeled.*

- *Content-based addressing:* A content-based addressing is an implicit addressing scheme whereby a message is delivered to nodes that are interested in its content. In this model, a consumer node may declare its interests by specifying some content. Then, when a producer node sends a message with content (or a special content-descriptor header) matching that specification, the network delivers the message from the producer to that consumer and possibly to others interested in the same content. There are several different ways to specify and therefore to "address" content. A hierarchical name with prefix matching is a simple form of content addressing. A "flat" name with exact matching, which corresponds to an explicit addressing scheme, can also be seen as a special case of content-based addressing. More elaborate forms of content addressing include content specifications in the form of a sort of query. Carzaniga et al. [24] propose a language for such queries to describe information within their notion of *content-based networking.* A conceptually similar but also simpler way of describing and addressing information is to use sets of tags.

1.4.1 Communication Models

Communication models or service models define different delivery services that a network might provide to applications. In general, a sender node issues a *request* (or routing request) containing the message that needs to be delivered plus possibly additional information needed to perform the service, such as a destination address, a content descriptor, or other parameters that would determine one or more destination node or the mode of delivery. In the following we provide simple definitions of common communication models and their routing requests.

Broadcast Broadcast is one of the simplest form of communication in the network. In this model a routing request $R = (m, s)$ consists of a message

and the identifier of the sender. The message will be delivered to all nodes in the network (except s).

Unicast In *unicast* model the goal is to establish communication between any two nodes as sender and receiver. In this point-to-point communication model, each node has a unique identifier. When a node s issues a unicast request $R = (m, s, t)$, the network has to deliver message m from source node s to the destination address t.

Multicast In *multicast*, a sender can send a message to several receivers in one single transmission. In this addressing scheme, each node can announce itself as a member of a multicast group and join the group. A sender sends its message to a multicast group and network nodes replicate the message if necessary and send it to all group members. Here a multicast request is in this form: $R = (m, s, groupID)$.

For the sake of simplicity, sometimes we may assume that the sender knows the address of all the multiple destinations for a message m. The sender can therefore issue a multicast request $R = (m, s, T)$ that, addition to the message content and source identifier, carries the list of destination $T = \{t_1, t_2, \ldots, t_k\}$.

Anycast the anycast model is similar to the multicast model, except that the network can satisfy an anycast request by delivering the message to at least one of the members of the group.

Notice that when the addressing scheme is implicit, such as with content-based addressing, the communication model can be any or all of the models listed above.

1.4.2 Network Models

The way nodes are interconnected in a network greatly influences the properties of the network for the purpose of routing. Depending on the network model, additional information about the network might be available and more or less structured. This includes link costs, node or edge capacities, single or bidirectional communication channels, etc.. Below we briefly describe two categories of networks that are the main focus of our research, namely *geographic networks* and *Internet-like* networks:

Geographic Networks

Nodes of the network are situated in a Euclidean space, and the distances between nodes are Euclidean distances (based on the positions of the nodes). One important property of distances in a geographic network is that they obey the triangle inequality: if three edges form a triangle in the network then the sum of the lengths of any two edges of this triangle is always greater than the length of the third edge. One can infer from this inequality that a direct line between two nodes is also the shortest path between them. There are different ways to establish communication between nodes of such network. *Unit-disk* communication model is often used in conjunction with geographic networks to represent wireless ad-hoc networks.

In the unit-disk model, we assume that every node has a communication range equal to one unit, such that two nodes located within one unit of each other can communicate directly. Unit-disk communication is often used to model wireless ad-hoc networks in which every node is assumed to be equipped with a wireless communication device that has a symmetric and limited range of communication. In Chapter 4, we study a special variant of geographic networks that uses *unit-disk* as the communication model.

Internet-Like Networks

Internet-like networks, as the name suggests, have a connectivity structure that is very similar to the structure of the Internet. The Internet is an interconnection of many administratively independent and often competing entities called *Autonomous Systems (AS)*. For commercial reasons, very few autonomous systems reveal their internal structure publicly. ASes typically also conceal their external interconnections and their routing policies, although some connectivity information can be inferred from global routing information. All of this is to say that it is very difficult if not impossible to determine the exact detailed structure of the Internet. Still, this structure has been studied extensively, mainly through large collections of traced routes, or through BGP or IGMP data. As a result, there Internet can be mapped quite accurately at the high-level (AS), and in general it can also be characterized in terms of a number of connectivity properties.

1.4.3 Characteristics of Routing Algorithms

A routing scheme consists of a set of algorithms that induce transmission paths through the network. Below are the main features and complexity measures of

a routing scheme.

Memory Requirements

Memory usage is an important characteristic of any routing scheme. The memory requirement of a routing algorithm can be measured in different ways. One metric is the average memory required at each node of the network. Another one is the maximum amount of memory needed by the algorithm at any given node. Sometimes, for the sake of comparison between different routing algorithms, we use the average of the memory consumption over all nodes of the network. Other times it is better to consider the distribution of memory requirements over all nodes, and more specifically some indicative quantiles, including the maximum.

Path Stretch

The stretch induced by a routing scheme S for the path between two nodes u and v is the ratio between the length of the path connecting u and v as defined by S, and the distance between u and v in the network graph G (that is, the minimal path length between u and v in G).

$$\mathsf{stretch}(u, v) = \frac{|route_S(u, v)|}{dist_G(u, v)}$$

More generally, given a route request r in a network G, a routing scheme S induces a stretch

$$\mathsf{stretch}_{S,G}(r) = \frac{|route_S(G, r)|}{|route_{\mathrm{OPT}}(G, r)|}$$

Where $route_{\mathrm{OPT}}(r)$ is an optimal routing of request r on G, which depends on the network model and communication scheme. For example in the case of unicast communication in general networks, an optimal route corresponds to a minimal path between source and destination, while for multicast communication an optimal routing is a minimal Steiner tree between the nodes that are involved in the multicast request.

The stretch of a routing scheme for a given network G is the maximum stretch of any allowable routing request in G.

Congestion

Every network element (node or edge) has limited resources that it can allocate to routing tasks. When the routing scheme puts demands for resources

that exceed the limits of one or more element, we say that the network and specifically those elements are congested. Typically, there are two important resources that are subject to congestion: link capacity and node capacity. Link capacity is the maximum amount of information a link in a network can carry at any instance of time. Depending on the network model, links or edges in the network may have uniform or varying capacities. Node capacity is very similar, specifically it is the maximum amount of information a node can process at any instance of time.

Adaptive Vs. Oblivious Schemes

A routing algorithm may use knowledge of previously routed requests to influence its current routing decisions. This is what we call an *adaptive* routing scheme. By incorporating this additional information in the decision-making process, an adaptive routing scheme might perform better for load balancing and to avoid network problems such as deadlock and livelock. In the opposite case—when routing decisions do not take into account previous requests in any way—we refer to such schemes as non-adaptive or *oblivious* routing.

Dynamic Vs. Static Schemes

A *static* routing scheme uses only network information that does not change in time, while a *dynamic* one uses the current state of the network. This dynamism can be further characterized depending on the type of information undergoing changes. In particular, we distinguish topological and non-topological changes. As an example of non-topological changes, we may have a static multicast scheme in which multicast groups are fixed, and conversely a dynamic multicast scheme in which group membership may change over time. In this document, we focus on non-topological dynamism in the network. In other words, we assume that topological changes are either non-existent or slow enough not to be a concern for our routing scheme.

Compact Routing: Memory Vs. Stretch

In the unicast model, a trivial solution for routing on the shortest paths between any two nodes would be to store, at every node v, the next hop of all all-pairs shortest paths, which means the next hop on each of the shortest paths from v to all other $n-1$ nodes. This solution, which is fully deterministic and optimal in terms of path lengths, requires $O(n \log n)$ bits of memory in each node of

the network. However, this solution is not good enough for large networks in which nodes have limited resources.

One can imagine other trivial unicast routing schemes. For example, a probabilistic one would be that each node v delivers the messages addressed to v, and otherwise forwards every other message at random. In this case, each node requires that each node v stores its own identifier, with $O(\log n)$ bits, and nothing else. So, this scheme would be extremely compact, but at the same time it would also be extremely inefficient in terms of path lengths, since the path lengths induced by the scheme (in expectation) would be much worse than the optimal.

Research on compact routing has focused almost exclusively on the unicast model. A notable exception is the work by by Abraham et al. on compact *multicast* routing [6]. The several compact algorithms proposed by Abraham et al. are very interesting, but they are not very practical, in the sense that they pose extreme trade-offs between memory and stretch. For example, one of the proposed algorithm is a name-dependent multicast routing scheme whose memory requirement at each node is $\tilde{O}(n^{1/k})$, uses labels of size $\tilde{O}(n^{1/k})$, employs headers of size $\tilde{O}(n^{1/k})$, whose stretch is $4k-2$.[5] Another algorithms is a name-independent multicast routing scheme whose total memory is $\tilde{O}(kn^{1+1/k}\log\Delta)$ with stretch $O(k)$.

Therefore, a new generation of routing schemes for the unicast model have been studied during the previous three decades under the general title of "compact routing". In brief, research in compact routing studies the fundamental limits of routing scalability, as well as algorithms that try to reach those limits. In another words, compact routing is the investigation of the necessary trade-off between routing-table sizes and routing stretch.

In general, compact routing schemes are divided into two categories: name dependent and name independent schemes. This categorization is based on the respective addressing scheme they require. Here we briefly summarize the research achievements in each category for the unicast communication model.

Name-dependent routing This category includes optimal compact routing schemes for general graphs and other restricted forms of graphs. Cowen [28] introduced the first non-trivial scheme with stretch 3 which needed $\tilde{O}(n^{2/3})$ memory. Thorup and Zwick [79] provided the best known stretch-3 name-dependent routing scheme with only $\tilde{O}(\sqrt{n})$ memory. Thorup and Zwick also generalize their scheme so that it achieves stretch $4k-5$ (and even $2k-1$ with handshaking) with $\tilde{O}(N^{1/k})$ memory, and also

[5] Soft-O notation: $f(n) = \tilde{O}(g(n))$ if $f(n) = O(g(n)\log^k(n))$ for some k.

introduce an optimal name-dependent routing schemes in trees that requires $\tilde{O}(1)$ memory and $\tilde{O}(1)$ header size (Fraigniaud and Gavoille [30] obtained the same result independently). Because of its compactness, this scheme became a building block for many other compact routing schemes [79, 4, 3, 1, 53, 9].

Name-Independent Routing This category started with the work of Awerbuch et al. [11]. Awerbuch and Peleg [12] were the first to show that constant stretch is possible with $o(n)$ memory per node, albeit with a large constant. After a gradual improvement on bounds, finally, Abraham et al. [4] obtained an optimal, stretch-3 routing scheme that uses $\tilde{O}(\sqrt{n})$ memory for a weighted undirected network. In another work [3], Abraham et al. showed that any name-independent routing scheme (even for single source) with maximum stretch strictly less than $2k+1$ requires $\Omega((n \log n)^{1/k})$ bit routing table.

1.4.4 Compact Routing: Memory Vs. Congestion

That of the memory requirements is the first problem that researchers faced in the scalability analysis of any routing scheme. However, with today's technological advances, increasing the speed and memory capacity of individual nodes in the network is not as problematic as it used to be. In a large-scale distributed system such as Internet, the connection bandwidth between nodes of the network is usually the primary performance bottleneck. In other words, the problem is congestion rather than memory. Of the few works that have studied the theoretical boundaries of the congestion problem, the most prominent one is the work by Harald Räcke [70]. Räcke developed a hierarchical decomposition of the graph into a distribution of trees that can be used in a probabilistic routing scheme in which each packet is randomly assigned to, and therefore routed along one tree. With such distribution of trees, the scheme achieves optimal congestion for unknown traffic in general graphs. However, this method is hardly practical, since it produces $O(E)$ trees, that is, a number of trees proportional to the number of links in the network.

1.5 Structure of the Document

The rest of this thesis is organized as follows: In Chapter 2 we present a tree-based routing algorithm for our tag-based information centric networking architecture. In Chapter 3 we then present a fast forwarding algorithm that

combines CPU and GPU processing units to achieve high-throughput in a tag-based ICN. Then in Chapter 4 we describe an oblivious low-congestion multicast routing scheme for geographic, wireless networks. We conclude in Chapter 5 with a short summary of the work and a few ideas for future work.

Chapter 2

Scalable Routing for Tag-Based Information-Centric Networking

In this chapter we focus our attention on information-centric networking, and in particular on routing. Recall that information centric networking (ICN) is a network architecture conceived and designed to allow applications to address information rather than hosts. In spite of the voluminous literature covering many different aspects of information-centric networking, routing remains an open problem, in particular with regard to the issue of scalability.

In its early incarnation that quickly became mainstream, namely the CCN/NDN architecture, ICN was designed to remain compatible with, and therefore supported on top of traditional IP routing. Specifically, the addressing in CCN/NDN is defined on *names*, and routing is based on name prefixes analogous to IP prefixes. However, while IP prefixes are network-defined, and therefore guarantee a basic level of aggregation when mapped onto the hierarchical structure of the network, the same can not be said of application-defined name prefixes. In essence, a fundamental problem is that the number of name prefixes would grow too large.

A related problem in the CCN/NDN architecture is the use of per-packet in-network state as an integral part of the routing scheme to cut loops and return data to consumers. This network state also hinders scalability.

In this chapter, we develop a routing scheme that solves these problems. The service model of our information-centric network supports information pull and push using tag sets as information descriptors. Within this service model, we propose a routing scheme that supports forwarding along multiple loop-free

paths, aggregates addresses for scalability, does not require per-packet network state, and leads to near-optimal paths on average. We evaluate the scalability of our routing scheme, both in terms of memory and computational complexity, on the full Internet AS-level topology and on the internal networks of representative ASes using realistic distributions of content and users extrapolated from traces of popular applications. For example, a population of 500 million users requires a routing information base of 3.8GB with an almost flat growth. We conclude that information-centric networking is feasible, even with (or perhaps thanks to) addresses consisting of expressive content descriptors.

2.1 A New Perspective on Routing in ICN

A fundamental problem remains open in information-centric networking: there is yet no demonstrably scalable scheme that supports true routing, that is, *packet switching* with multiple sources and destinations, as opposed to a per-flow or per-object lookup followed by a traditional host-based (i.e., location-based) data transfer. In fact, largely because of this gap, the validity and utility of a content-centric network layer has been rightly called into question [32].

The primary approach to routing and forwarding in ICN (as typified by CCN/NDN [41], but also in the earlier work on TRIAD [33]) is to adapt IP routing to use name prefixes instead of IP prefixes. While this approach has the great advantage of reusing much of the current network infrastructure, it also has fundamental limitations. First, since it is based on traditional *unicast* routing, it cannot reliably support multiple sources or destinations for the same information. A router may list multiple next hops for the same prefix, but the routing scheme provides no indication of how to forward consistently across routers so as to follow one *path* to a destination (or multiple paths to multiple destinations). Moreover, multiple next hops may lead to loops. In fact, the main approach is not to avoid loops, but merely to detect them, tracing each packet throughout the network with per-packet state, thereby increasing the overall cost of forwarding. For analogous reasons, unicast routing/forwarding cannot directly support "push" ICN communication [23]. Here again, the already vast and growing content space is believed to pose a fundamental scalability limitation to traditional routing.

We develop a different approach to routing, one based on *trees* in which edges are annotated with *content descriptors*. This new routing scheme has the following novel and important properties:

- It is compatible with in-network caching, as well as the full range of ex-

isting ICN addressing schemes, from content identifiers [47] to structured names [41] to tag sets [22]. We choose tag sets, since they are strictly the most expressive form of descriptor and yet admit to an intuitive and effective aggregation that is fundamentally superior to the aggregation of, say, name prefixes.

- It provides loop-free paths to multiple destinations, meaning that communication can be dynamically assigned an arbitrary fan out, from anycast (forward to any one of many destinations) to m-anycast (at least or at most any m destinations) to multicast (all destinations).

- It provides extremely compact and efficient *locators* that can be used to achieve the throughput of current networks within the content-centric service interface.

- It does not require the presence of per-packet soft state within the network, unlike previous designs.

Now, a single tree may not use the most direct paths and would be more vulnerable to congestion and network partitioning. We therefore use *multiple trees* so as to reduce path lengths on average, reduce congestion, and improve reliability. We develop a hierarchical multi-tree routing scheme that allows for the creation of sets of trees with specific properties at different levels (e.g., shortest-paths trees within an AS along with policy-specific inter-AS trees).

In principle, however, multiple trees also require larger routing tables, which leads us back to the fundamental question of scalability. We address the issue of scalability through the aggressive aggregation of content descriptors. Beyond the natural aggregation of tag sets, we develop a routing table based on PATRICIA tries that aggregate content descriptors *across all trees*.

We evaluate the memory complexity of the routing scheme and its implementation at the global network scale. We emulate the scheme over the full AS-level topology of the current Internet and within a number of representative ASes. In order to test the scheme under realistic current and potential future application demands, we extrapolate from traces of some characteristic content-driven applications [22]. These extrapolations give us various workloads of content descriptors that correspond to several hundred million users. We then use such workloads to assess the concrete memory requirements of the scheme on routers at the local (intra-AS) and global (inter-AS) levels.

Our analysis shows that content descriptors indeed aggregate effectively and, therefore, the routing information base remains contained in size even

with a growing population of users and, consequently, more and more content descriptors. For example, for a number of representative applications, a population of 500 million users using a total of nearly 10 billion content descriptors would require a routing information base of 3.8GB, with an almost flat growth for additional users enabled by effective aggregation.

Here we introduce locators and content identifiers, we detail the scheme over a hierarchy of domains, and we develop concrete data structures to represent and aggregate routing information for which my colleague Michele Papalini has also developed incremental update and maintenance algorithms [65, 64]. We also conduct an extensive and in-depth analysis of the scalability of the scheme.

2.2 Network Architecture

We begin by describing the service model, addressing scheme, and architecture of our information-centric network. The service model extends our prior ICN design [22, 23] and is also a significant superset of other, related models [41, 47]. We review the basic model here for clarity and completeness. We also introduce two extensions to the network architecture not previously described, namely *locators* and *identifiers*.

The request/reply service consists of three primitive network functions:

Offer: A producer registers one or more descriptors that identify the data that the producer are willing and able to provide.

Request: A consumer requests data by issuing a request packet carrying a content descriptor or a content locator (detailed below). The network then delivers the request packet to one or more producers that are willing and able to satisfy the consumer's request.

Reply: A producer (or a caching router) responds to a request packet by returning a reply packet carrying the requested data.

The request/reply service define and use routing information consisting of *content descriptors*. Here the producers define routing information that attracts request packets towards them.

Both request/reply and publish/subscribe services define and use routing information consisting of *content descriptors*. In request/reply, producers define routing information that attracts request packets towards them, while in publish/subscribe it is consumers that define routing information to attract

notifications. Thus, routing for the two services differ only in the sources of routing information, but is otherwise conceptually identical. We therefore propose a network interface with a single *register* function to define routing information.

The semantics of descriptors is also identical for requests and notifications, which means that the matching algorithm used for forwarding requests and notifications is the same. However, the treatment of the two packets differ in other ways. A request is ideally an anycast packet, while notifications are multicast. Also, a request is expected to generate a corresponding reply, while a notification is a one-way message. Furthermore, the caching semantics are different. A request that can be satisfied by cached content will not be forwarded downstream toward the original producer, while a notification must be forwarded all the way to interested consumers (although notifications might also be cached for reliability purposes).

The network also defines opaque host *locators*. Locators are attached to requests so that the corresponding replies can be forwarded back to the requesting application. In addition to replies, we propose to use locators to forward requests. In particular, a data reply can also carry the locator of the producer so that the consumer can address follow-up requests (e.g., for the next data blocks) directly to that producer. Locators may be implemented with stable unicast addresses (e.g., IP addresses) or they may be based on transient state (e.g., a nonce that identifies a trail of pending interests in CCN). In Section 2.3.2 we detail an extremely efficient form of locators usable within our routing scheme.

2.2.1 Content Descriptors

Descriptors play a central role analogous to IP prefixes. The semantics of descriptors define the semantics of the network service, and in particular they define how data replies match requests, how offers match requests (and, therefore, how offers describe the data available from a producer), and how notifications match subscriptions. As discussed so far, descriptors are abstract and generic. Indeed, much of what we propose is conceptually independent of their specific form and semantics. However, in order to develop a concrete service and a corresponding concrete routing scheme, we must define descriptors. For this purpose we adopt “tags”.

A descriptor consists of a set of string tags, with the matching relations corresponding to the subset relations between sets of tags: a descriptor R in a request would match a descriptor O in an offer when the request con-

tains all the tags of the offer ($R \supseteq O$). Consistently, a descriptor N in a notification would match a descriptor S in a subscription when the notification contains all the tags of the subscription ($N \supseteq S$). For example, an offer for $\{icn14, paper\}$ would match a request $\{paper, routing, icn14\}$ or a request $\{icn14, paper, pdf, n32\}$.

Notice, however, that tag sets are strictly more expressive than name prefixes. A name prefix can be represented as a single tag set. For example, */org/gnu/software/* can be written as tag set {*1:org*, *2:gnu*, *3:software*}, and would match descriptor {*1:org*, *2:gnu*, *3:software*, *4:emacs*}. Conversely, the semantics of a tag set would require exponentially many prefixes (all permutations) to express the same descriptor.

Tag sets aggregate analogously to prefixes. In particular, a descriptor X subsumes all other descriptors Y that contain X. For example, any descriptor matching $\{music, jazz\}$ would also match its subset $\{music\}$, so a router might combine the two by storing only the more general tag set $\{music\}$. We discuss more about aggregation in Section 2.3.6.

Tag sets differ from IP prefixes in a crucial way. While IP prefixes are assigned by network designers and administrators, descriptors are assigned by applications. This is also true of other forms of addressing in ICN, including names in a hierarchical name space or flat identifiers. In fact, application-defined addressing is arguably the most important defining property of ICN. Allowing applications to define network addresses empowers applications but at the same time leaves the network and applications themselves vulnerable to conflicts and also abuses in the use of the address space. With tag sets, as for name prefixes, this problem can be greatly reduced through conventions, for example by defining reserved tags and mandatory scoping tags equivalent to host names in URLs.

Content Identifiers

A content descriptor (a tag set) may contain a unique identifier, such as a cryptographic hash of the content, or an object identifier plus a version number and a block number. In this way, a descriptor can identify a data block uniquely. More generally, tag sets can encode meta-data and higher-level protocol information. However, we believe that information that has a specific function at the network or transport level should be represented with specific headers. In particular, at the network level we propose to use cache-control headers, as well as a *content identifier* to refer to a specific data block. The form of such identifiers is defined at higher levels, for example to allow a transport protocol

to refer to the next sequence of blocks within a stream.

This separation between descriptors, identifiers, and other headers is consistent with the design of a protocol such as HTTP, where the URI does not identify a piece of immutable content, but other headers can be used for that purpose (e.g., ETags, Modified-Since) and yet other headers can specify additional properties of requests and replies, such as cache controls. This design also allows us to represent descriptors using a compressed, fixed-width header that hides individual tags. We describe this compressed representation next.

Representation of Tag Sets

Conceptually, a descriptor is a set of tags. Concretely, we represent descriptors as Bloom filters, and we develop our routing scheme around this representation. So, packets and routing messages carry Bloom filters, and the aggregation of routing information applies equivalently to them. Matching two descriptors amounts to checking the inclusion relation (bitwise) between two Bloom filters, while matching a descriptor against a predicate (i.e., a set of descriptors) amounts to finding one or more Bloom filters in the predicate that are subsets (bitwise) of the input Bloom filter.

In order to choose good Bloom filter parameters, which must be global properties of the routing scheme, we conservatively estimate here that tag sets would most likely contain no more than 15 tags. We therefore use Bloom filters with $k = 7$ hash functions and $m = 192$ bits, which ensures that a subset test $S_1 \subseteq S_2$ would be accurate up to a false-positive probability of $(1-e^{-k|S_2|/m})^{k|S_1 \setminus S_2|}$. For example, for a descriptor of $|S_2| = 10$ tags, a test $S_1 \subseteq S_2$ with another descriptor S_1 that differs by $|S_1 \setminus S_2| = 3$ tags would evaluate to true (a false positive, since $|S_1 \setminus S_2| > 0$) with probability 10^{-11}. Of course, these are network configuration parameters that can be set as appropriate.

2.3 Routing Scheme

We introduce a routing scheme based on multiple trees. At the core of the scheme is content-based routing on a spanning tree. We enhance this basic scheme with locators and with multiple trees within routing domains and over a hierarchy of domains (intra/inter-AS).

2.3.1 ICN Routing on One Tree

Consider a network spanned by a tree T. For now consider a router-level network. T is identified within each notification and request packet so that each router v can determine the set adj_v^T of its neighbors that are also adjacent to v in T. This can be done by adding an identifier for T in the packet and storing the adjacency set adj_v^T at each router v.

The forwarding information base (FIB) of router v associates each neighbor w in adj_v^T with the union $P_{T,w}$ of the predicates registered by all the hosts reachable through neighbor w on T, including w. Figure 2.1 shows an example.

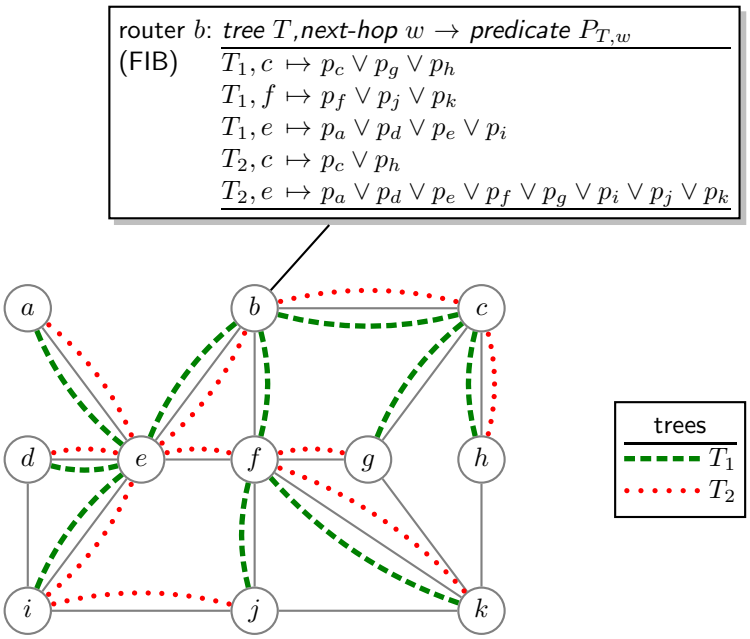

Figure 2.1. Multi-Tree Routing Scheme.

With a FIB representing $P_{T,w}$ for all neighbor routers w in adj_v^T, forwarding proceeds as follows: Router v forwards a packet (notification or request) with descriptor X received from neighbor u on tree T to all neighbors $w \neq u$ in adj_v^T whose associated predicate $P_{T,w}$ matches X. We say that a predicate P matches a descriptor X if one of the descriptors in P matches X.

Since we use trees, we can control the global fan-out of a packet with local decisions. A packet starts with its global fan-out limit k set by the sender. A limit of $k = 1$, which is the default for requests, corresponds to an *anycast* delivery (the network delivers one copy of the packet), while a limit $k = \infty$, which is the default for notifications, corresponds to a *multicast* forwarding

(the network delivers as many copies as there are interested receivers). A limit $1 < k < \infty$ can be also used and requires only minimal additional local processing: the router selects at most k matching neighbors and then partitions the fan-out limit over the selected neighbors. This guarantees the delivery of the packet to at most k destination. If the router does not partition the fan-out limit over the selected neighbors and instead always send k as the fan-out limit along the way, then the packet will be delivered to at least k destinations.

2.3.2 Unicast Routing on Trees

We combine a locator-based unicast routing service with descriptor-based routing. Here we provide an overview of the labeling and forwarding scheme that we use for locators. Labeling is the process by which the network assigns locators and locator FIBs to nodes. In their seminal paper on stretch-3 compact routing scheme for general graphs [79], Thorup and Zwick also propose a lesser-known but still practical compact routing scheme for trees. Since we use trees as a basic routing structure, we adopt this Thorup and Zwick scheme for locator-based forwarding and in particular we refer to the implementation of our locators as TZ-labels. Here we review the scheme only very briefly, without going through the details of the labeling algorithm and its associated forwarding algorithm, and we refer the reader to Section 2 of Thorup and Zwick. Using this scheme, given the TZ-label of the destination plus its own TZ-label, a router can compute the next-hop towards the destination.

This scheme is extremely efficient both in space and time. In terms of space, a router needs to store its own TZ-label, which is at most $(1 + o(1)) \log_2 n$-bit long for a network of n nodes.

The scheme can also be built efficiently. A tree can be labeled with a two-step distributed algorithm. In the first step, which could be combined with the construction of the tree, a converge-cast algorithm calculates the size of descendants of each node on the tree, while the second step consists of a depth-first-search numbering of nodes on the tree.

2.3.3 Locators and the Request/Reply Service

As discussed in Section 2.2, the network forwards packets using either an explicit destination locator or a content descriptor if no locator is given. Locators are network-defined quantities that may or may not have permanent validity (like IP addresses).

In our routing scheme we use TZ-labels to implement node locators. A node locator consists of a tree identifier plus the TZ-label of the destination on that tree. We now sketch a simple request/reply protocol that combines locators and descriptors, and that can be the basis for a full transport protocol for ICN.

The general idea is to use descriptors to find an object—that is, to forward a request towards a producer capable of satisfying the request—and then to use the more efficient locators to return the data block back to the consumer and also to request other data blocks from the same producer. To implement this idea, a request packet must carry the locator S of the source application (the consumer). When a request reaches a producer capable of satisfying the request or a router with a valid cached copy of the data, the producer or caching router sends back a data reply with destination locator S, which the network forwards back to the requesting application.

The advantage of locators within our scheme is that requests, unlike interest packets in CCN [41] which create a trail of pending interests, do not require any per-packet in-network state. Without per-router pending-interests tables, our scheme does not support the aggregation of simultaneous identical requests. However, identical requests that are not exactly simultaneous can still be effectively aggregated by caching data along the forwarding path.

A data reply may also specify one or more locators of the producer as its origin, as well as the identifiers of one or more follow-up data blocks. Specifically for our scheme, the multiple locators can be obtained using multiple trees, an approach that we detail below. A consumer receiving a data reply with an origin locator may then use that locator to send follow-up requests directly towards the same origin. This, in particular, can substantially reduce the overhead of transferring large files.

Locators built on TZ-labels are relatively stable, since they change only when trees are rebuilt, for example in response to a topology change. Still, locators may also change within a flow if producers or consumers move within the network. A transport protocol that intends to support such mobility must correctly switch locators as applications move.

Lastly, a limitation of TZ-labels is that they may reveal the identity of consumers. If anonymity is required, then locators should be based on an appropriate anonymity-preserving routing scheme, such as onion routing [36].

2.3.4 Using Multiple Trees

Routing on a tree has two disadvantages. First, paths might be "stretched", meaning the distance between two nodes on the tree might be longer than on the full graph. Second, traffic would flow only on the tree, reducing the overall network throughput. It is well known that these problems can be alleviated by using multiple trees, and therefore we extend our routing scheme to use multiple trees. A notification or request is committed to, and thereafter routed using, one of those trees. Therefore, the forwarding process is identical to that over a single tree for an individual request or notification, but traffic is more evenly distributed and path lengths shortened on average. However, two aspects of the multiple-tree scheme are non-trivial: how to build and then select trees, and then how to combine multiple trees at different levels in hierarchical routing.

Building and Selecting Trees

The key to increasing throughput and reducing path lengths is in the choice of trees: first, the routing process must produce a good set of trees; second, when a request or a notification enters a routing domain, the access router must assign the request or notification to a tree in that domain. The choice of trees, the way they are built and then assigned by routers, could also be used to implement various routing strategies and policies.

The problem of covering a network with trees so as to achieve specific design objectives has been studied extensively from a theoretical perspective. For example, Räcke formulated a method to cover a network with trees to achieve the theoretically minimal congestion under unknown traffic [70]. However, such results are not applicable in practice, primarily because they can require an extremely high number of trees.

Our approach to building and selecting trees is therefore based on heuristics. To date we have studied two such heuristics for global trees, which are arguably the most crucial, and one for local trees.

H1: Latency Only (L) We choose a small number of root ASes and then build a shortest-paths (Dijkstra) tree for each root AS. This heuristic is intended to favor latency over any other routing objective. For the purpose of the analysis, we use a uniform-random choice over all ASes, which should give more conservative results. In practice, root ASes can be chosen in a number of ways using a distributed leader-election algorithm, perhaps favoring higher-tier ASes. Another and perhaps better way to

select root ASes is to do it off line through a global administrative body, similar to the way top-level DNS servers and structures are configured today.

H2: Latency and Congestion (LC) We start with a first shortest-paths tree rooted at the AS with the lowest eccentricity representing the center of the network. We then increase the cost of each link used by the tree, and proceed iteratively to find another tree. The weight increase is by a fixed amount and, therefore, linear in the number of trees. At each iteration we select a new shortest-paths tree rooted at the AS with the lowest eccentricity. The new tree is computed with the current adjusted link weights and, therefore, it is likely to differ from all previous trees. These trees can be constructed using a slightly modified version of the fast distributed algorithm of Almeida et al. [8], which computes the eccentricity of node v in $diameter(G) + ecc(v) + 2$ rounds.

At the global level, trees are heavy in terms of memory because they store the aggregated predicates of the whole network. Therefore, we compute a relatively small number of global trees. Furthermore, we use shortest-paths trees that can be computed efficiently in a completely decentralized manner. Conversely, at the local level, trees are lighter and can be efficiently computed in a centralized manner. Since latency is crucial at the local level, the heuristic we use for local trees is also based on shortest-paths trees.

H3: Minimal Latency: We build shortest-paths trees rooted at every router within an AS.

To assign trees dynamically, routers select trees uniformly at random at the global level, while at the local level they always choose their own shortest-paths tree so as to obtain latency-minimal routes. In Section 2.4 we evaluate our scheme under the three heuristics.

2.3.5 Hierarchical Multi-Tree Routing

The routing scheme we propose can be extended over a hierarchy of levels within the network, with multiple trees eat each level. We describe the case of two levels (intra- and inter-AS), although the scheme generalizes to more levels.

Routes are defined by global trees that span the AS-level network, and by local trees that span the internal network of each AS. Conceptually, each tree

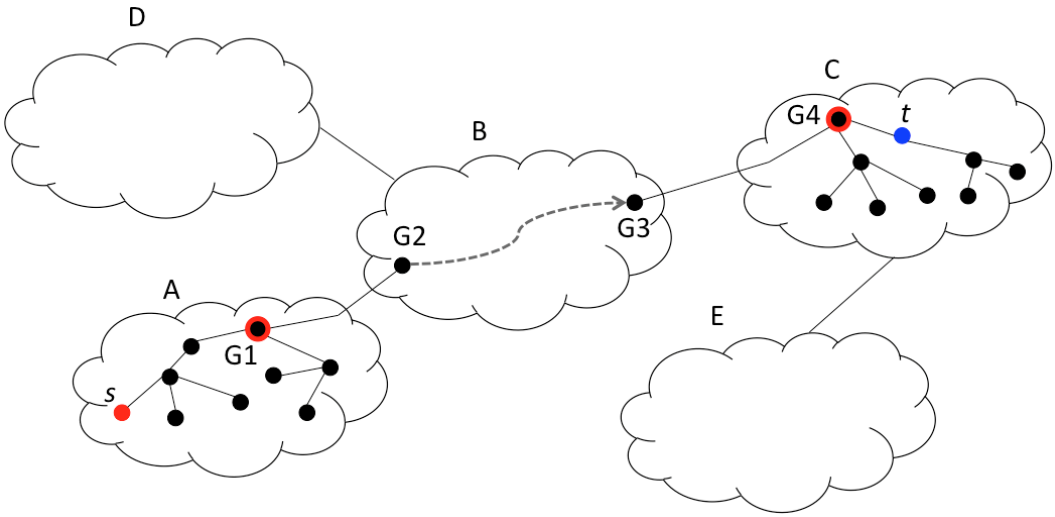

Figure 2.2. Hierarchical Routing: Inter-AS.

has a separate FIB, but concretely we aggregate predicates across trees so as to reduce space (as discussed below in Section 2.3.6). The FIB of a global tree contains the aggregate predicates of all the ASes. The FIB of a local tree contains the predicates of each internal host, possibly aggregated at the subnet level. An interior router needs to know only the local trees of its AS plus the TZ-labels of at least one gateway router for each global tree. A gateway router needs to know the local trees, the global trees, and the exterior connectivity of all the gateway routers of its AS, including their TZ-labels on the local trees. With this information, the network can forward packets based on either content descriptors or locators. We describe these two algorithms in turn.

Descriptor-Based Forwarding

A packet (request or notification) is first assigned to a local tree by its access router, and on that tree it is forwarded based on its content descriptor and fan-out limit as explained in Section 2.3.1. In addition to that, the packet is assigned to a global tree and sent to a gateway router that belongs to that tree using the TZ-label of that gateway on the local tree, which is known by the access router. On its global tree, a packet reaching a gateway router (or starting from that gateway) may have to cross the AS of that gateway to reach other gateways connected to the next-hop neighbor ASes on the global tree. This again is done on a local tree based on the TZ-labels of those gateways. And if the packet is entering that AS for the first time, then the local forwarding is

performed via the content descriptor.

Locator-Based Forwarding

In our hierarchical routing scheme, a locator consists of a stack of node locators, each one consisting of a pair (T, ℓ) where T is a tree identifier and ℓ is the TZ-label of the destination node on T. With two levels, a destination locator contains the node locator (T_{AS}, ℓ_{AS}) of the destination AS on an AS-level tree T_{AS} plus the node locator (T_r, ℓ_r) of the destination router r on an inter-AS tree. Given a destination $(T_{AS}, \ell_{AS})/(T_r, \ell_r)$, forwarding proceeds as follows: If already in the destination AS, the access router pops the (T_{AS}, ℓ_{AS}) locator from the locator stack and forwards the packet on tree T_r using TZ-label ℓ_r. Otherwise, the router pushes a locator (T, ℓ_g) of a gateway router of its AS using any intra-AS tree T, and then forwards the packet accordingly. When a packet reaches the destination at the top of the stack, the router pops the locator and proceeds with what is left on the stack. If the top locator is at the AS level, then the gateway router might have to cross its AS to reach another gateway, in which case it would push a locator of that gateway onto the stack.

Figure 2.2 shows an example for the two-level hierarchy described above. In this example, a node in AS network A has a content that is of interest for a node in AS network C. Node s is the access router of the producer of the content, and node t is the access router of the interested consumer. AS-level networks are interconnected within global trees. For the sake of simplicity, in Figure 2.2 we only show one global tree that connects AS-level networks with each other. and Inside each network, we only show one local tree. Gateway routers are marked with $G1, G2, G3, G4$.

2.3.6 RIB Representation

We now describe a concrete implementation of the routing information base (RIB) for the multi-tree routing scheme. Conceptually, the RIB of a router v stores the following information for each tree T:

- adj_v^T is the adjacency list of T at v, meaning the subset of v's neighbors adjacent to v on T.
- ℓ_v^T is the TZ-label of router v on T.
- $P_v^T : w \to P_{T,w}$ is a map that associates each neighbor w in adj_v^T with a predicate $P_{T,w}$, where $P_{T,w}$ consists of a set of content descriptors (see Section 2.3.1 and, in particular, Figure 2.1).

tree,next-hop	→ *predicate*
T_1,c	→ 00100101
	01010000
	01000001
T_1,f	→ 00100100
	01010000
T_1,e	→ 00010000
	10000101
T_2,c	→ 00100101
	01000001
T_2,e	→ 00010000
	10000101
	00100100

predicate	→ *(tree,next-hop)...*
00010000	→ (T_1,e), (T_2,e)
00100101	→ (T_1,c), (T_2,c)
10000101	→ (T_1,e), (T_2,e)
01010000	→ (T_1,c), (T_1,f)
01000001	→ (T_1,c), (T_2,c)
00100100	→ (T_1,f), (T_2,e)

Figure 2.3. RIB Indexed by Tag Set.

Our primary goal is to obtain a compact representation of the RIB that also allows for efficient incremental updates. adj_v^T and ℓ_v^T require minimal space and standard data structures, and are also stable with trees. The P_v^T map changes with changing application preferences (content descriptors) and is also by far the heaviest component of the RIB. We therefore focus on the implementation of P_v^T.

With a naive implementation (depicted in Figure 2.1), multiple trees would have completely independent predicate maps P_v^T with only the basic aggregation of descriptors (described in Section 2.2.1). However, trees are likely to share many descriptors, simply because the descriptors represent offers or subscriptions that must be reachable from all trees. This suggests a representation of the predicate maps that further compresses the routing information across trees. To exploit this form of aggregation, we develop a data structure in which routing information is not grouped by interface or tree, but rather by tag set. In practice, the RIB consists of a dictionary of tag sets, each associated with a set of tree-interface pairs. Figure 2.3 shows an example of this type of aggregation corresponding to the network of Figure 2.1.

We use a PATRICIA trie to index the Bloom filters representing the tag sets, and we associate each tag set with a table of 16-bit entries representing tree-interface pairs. An example is shown in Figure 2.4. PATRICIA tries have the advantage of requiring a minimal amount of memory, while also allowing for simple subset/superset checks implemented as tree walks. These checks are the essential building blocks for the maintenance of the RIB. The trie allows us to shortcut the search, much like a prefix search: if we are looking for subsets

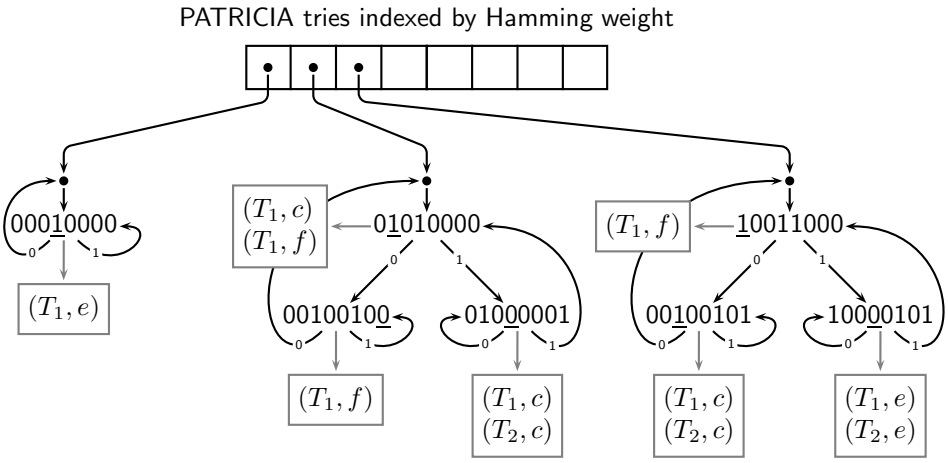

Figure 2.4. PATRICIA Trie Used for the RIB.

of an input filter f, and f contains a zero in a certain position identified by a node n, then we can skip the whole subtree of filters under n that contain a one in that position. In addition, we group filters by Hamming weight (in smaller tries). This allows us to skip entire tries containing filters that have too many elements to be subsets (or too few to be supersets) of the input filter. Since tries are independent of each other, subset/superset operations on different tries can also proceed in parallel.

Routing information propagates through update messages containing multiple descriptors, divided into an *addition delta* that is a set of filters to be added into the RIB, and a *removal delta* that is a set filters to be removed from the RIB. In the presence of dynamic, user-defined addresses in ICN network, an incremental maintenance algorithm is absolutely essential to keep the RIBs up to date. My colleague Michele Papalini has developed such maintenance algorithm that applies these update messages to perform incremental RIB updates [65].

2.3.7 Locator-Based Matching Algorithm

To forward a message toward a destination node (on a tree) each intermediate node needs only the TZ-label of the destination node (on that tree) plus its own TZ-label, which also serves as the node's FIB.

Algorithm 1 which is an adaptation of the algorithm described in Section 2 of Thorup and Zwick [79], indicates all the local variables that we need to store on each node. Thus each node stores its own label (my_label), which contains

Algorithm 1: Locator-Based Forwarding Algorithm.

```
struct TZ_label {
    uint16_t node_id;
    uint16_t ifx_list;
    uint16_t mask;
};
struct TZ_label my_label;
uint8_t k = leftmost_bit(my_label.mask);
uint16_t P[2] = {parent_interface, heavy_child_interface};
uint16_t f = largest_descendent_id;
uint16_t h = heavy_child_id;

int forward(struct TZ_label & dest) {
    v = dest.node_id;
    L = dest.ifx_list;
    M = dest.mask;
    return ((v >= my_label.node_id && v < h)
            ? (L >> k) & ((M >> k) ^ ((M >> k) -1))
            : P[v >=h && v <=f]);
}
```

the node identifier (node_id), a list of interfaces encoded in a bit string (ifx_list), and a mask used to extract them (mask). Each node also stores a constant k that indicates the size of the local mask (in bits), the identifier f of the largest descendant, and the identifier h indicating its heavy child. Heavy child is the child through which it is possible to reach the majority of the descendants. In addition, a node stores a vector P that contains the interfaces where to forward packets for the parent node and the heavy child.

The forward function extracts all the information needed from the incoming TZ-label (dest), and returns the output interface. Notice that this forwarding decision is taken in a single line of code that amounts to a handful of machine instructions.

2.4 Evaluation

We now present the results of an extensive experimental evaluation of our ICN routing scheme. We first evaluate the performance of the TZ forwarder; we then assess the *effectiveness* of the scheme in routing information over the Internet using a few trees; and after that we study the *scalability* of the scheme

both in terms of the memory requirements posed on routers and also in terms of the cost of maintaining routing information for large numbers of content descriptors.

A crucial difficulty in conducting this analysis is that there is no known deployment of an information-centric network at the scale we are targeting, therefore we use synthetic workloads, which we also detail here in Section 2.4.3. We conduct our analysis on the Internet AS-level topology compiled and maintained at UCLA's UCLA Internet Research Lab, consisting of a graph of 42,113 nodes and 118,040 edges.[1].

2.4.1 Speed of Unicast Forwarding Using TZ-labels

Using Thorup and Zwick labeling scheme, each TZ-labels computed for the AS-level network topology is at most 46-bits long, therefore, we decided to allocate 64 bits in the packet header. These 64 bits encode the label presented in the TZ_label structure in Algorithm 1. As you can see in this algorithm, forwarding decision is very simple and is essentially a single line of code, which amounts to a handful of machine instructions. To evaluate the performance of this algorithm, we generated a traffic consisting of unicast requests between every two nodes in the network. Then we measured the performance of every node in the network under this traffic and took the average. On average, forwarding decision of the TZ-matcher took 10 CPU cycles and achieved a throughput of 250M packets per second on a general-purpose, commodity CPU.

2.4.2 Effectiveness with k Trees

Here we consider the topological aspects of routing, and more specifically we evaluate the ability of our scheme to use the underlying network effectively. We use two measures of cost: *stretch* and *congestion*.

Stretch is the factor by which the distance between two nodes is extended by the routing scheme. Since our scheme routes each packet on a tree, this is the ratio between the distance on the tree and the distance on the full graph. For example, in Figure 2.5 the distance between node i and node j is 4 on the green spanning tree while these nodes are adjacent in the original graph. This means that the path between node i and node j is stretched 4 times. Given a set of k trees, the stretch for the path between two nodes is the expected

[1]Internet AS-level topology archive. Data retrieved 29/06/2012. http://irl.cs.ucla.edu/topology/

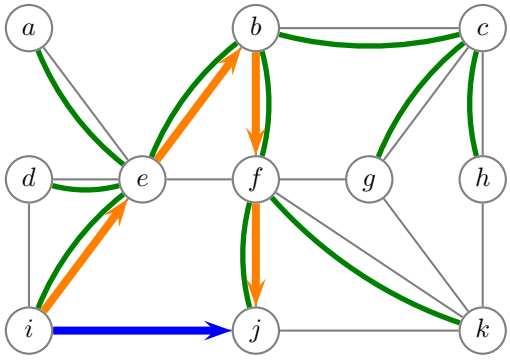

Figure 2.5. Stretch problem: $stretch_T(i,j) = \dfrac{distance_T(i,j)}{distance_G(i,j)} = 4$

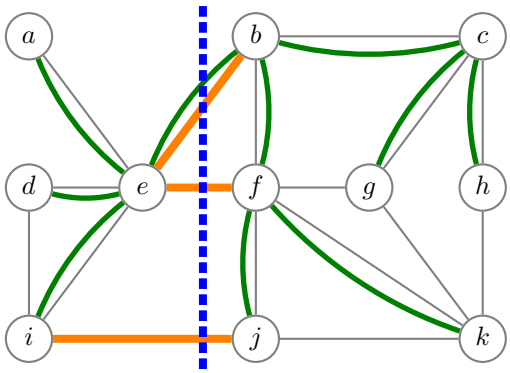

Figure 2.6. Congestion problem: $congestion_T(e,b) = load_T(e,b) = 3$

stretch; since we choose trees uniformly at random, it is simply the average stretch.

Congestion is the factor by which the usage of a link would grow using the routing scheme as compared to an optimal usage of the full network graph. The optimal usage here refers to the link usage with a distribution of traffic that achieves the best possible throughput. In practice, for each tree T, given a link (u,v) in T, we compute the cut defined by that link on T, meaning the partition of the nodes that are on the two sides of the link on T. We then compute the number of links that cross the cut on the original graph, which is the total capacity of the network over that cut. For example in Figure 2.6, the edge (e,b) on the green spanning tree defines a cut that splits the nodes of

the tree into two partitions. On the original graph, three edges of (e, b), (e, f) and (i, j) connect these two partitions together, hence the $load_T(e, b) = 3$. We assume that, for the portion of traffic routed on T ($1/k$ of the total traffic for k trees), the link (u, v) would need to carry the traffic that could instead go over all the links that cross the cut. So, for a cut of size $s_{T,u,v}$ on a tree T out of k trees, link (u, v) is given a congestion of $s_{T,u,v}/k$, and the total congestion of that link is the sum of its congestion for all the k trees. Notice that this congestion factor is a very conservative measure, since it uses the globally optimal allocation of flow for all network cuts as a baseline.

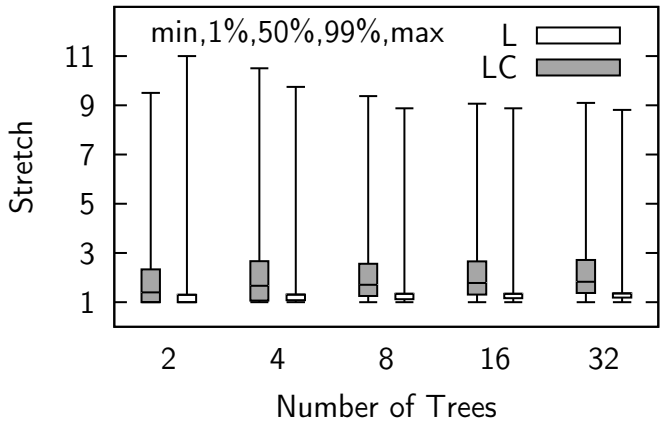

Figure 2.7. Path Stretch.

In Figure 2.7 we show the expected stretch for various sets of global (AS-level) trees. We generate sets of 2, 4, 8, 16, and 32 trees using the heuristics H1 and H2 discussed in Section 2.3.4. The label L in the plots refers to the latency-only heuristic, H1, while LC refers to the latency-and-congestion heuristic, H2. Each box plot in the chart shows the minimum, the 1-percentile, the median, the 99-percentile, and the maximum. The plot shows that the maximum expected stretch decreases with more trees, while more trees lead to a minor increase of the median (expected) stretch. Despite the growth, we can see that the stretch is low: the median always remains under 2 and the 99-percentile under 3. There is also a clear difference between the two heuristics: heuristic L achieves better results than LC.

Our experimental analysis is consistent with another study on the approximability of the AS-level topology with trees. Krioukov et al. [49] studied various compact routing schemes with a theoretical expected stretch of 3 [28, 79], and found that in practice, on two AS-level topologies measured by the Skitter

and DIMES tools, respectively, their average stretch is instead very close to 1. The result of their study is shown in Table 2.1.

	average stretch		
	Skitter	DIMES	
scheme	(9204 nodes)	(13931 nodes)	**number of trees**
TZ	1.08	1.13	$\tilde{O}(n^{1/2})$
BC	1.06	1.03	$\tilde{O}(n^{2/3})$
TZ/BC hybrid	1.02	1.01	$\tilde{O}(n^{2/3})$
Abraham	1.35	1.45	$\tilde{O}(n^{1/2})$

Table 2.1. Stretch of Compact Routing Schemes in Practice.

These routing schemes in practice provide near optimal routing paths by using quite a large number of trees. The routing scheme we described in this chapter also achieves an average stretch very close to 1, but with significantly smaller sets of trees. Note that the compact routing schemes that were the subject of Krioukov's study, guarantee a theoretical upper bound for the stretch of paths. Our routing scheme does not provide this guarantee and hence it is able to achieve near optimal average stretch with only a constant number of trees for the AS-level topology network.

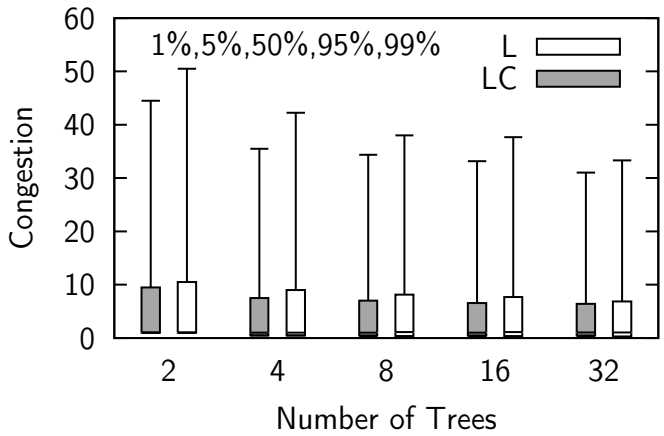

Figure 2.8. Link Congestion.

Figure 2.8 shows the congestion for the same set of trees of Figure 2.7. This plot shows the 1-percentile, 5-percentile, median, 95-percentile, and 99-percentile of the distribution. The salient result is that most links experience no congestion penalty at all, experiencing a congestion factor of 1, and further that

extreme levels of congestion are reduced when using more trees. As expected, the congestion factors for the L heuristic are higher as compared with the LC heuristic.

The analysis of stretch and congestion shows that different tree-building strategies may be used to achieve different design goals. More importantly, the general conclusion we can draw from this analysis is that even small sets of trees can cover the Internet at the AS-level topology quite well, with only minimal cost in terms of path-length stretch and link congestion.

2.4.3 Application Workloads

Our objective is to create workloads corresponding to the *plausible* behavior of applications over a global-scale information-centric network. To do that, we build models of future applications by extrapolating the behavior of existing applications (and their users) for which we have significant real traces.

Here we are only interested in the part of such workloads that are relevant for routing, namely (1) content descriptors used in offers issued by information producers in "pull" information flows, and (2) content descriptors used in subscriptions issued by consumers in "push" flows. We consider four classes of applications: (1) "pushing" generic Web content and blog posts; (2) "pulling" video content; (3) "pushing" short messages and following short-message publishers; and (4) "pulling" BitTorrent. We now discuss each class of application and the corresponding network workload.

Active Web. We envision a future information-centric network used to actively distribute Web content. Rather than analyzing traditional Web requests in terms of access to individual servers, we try to understand what users are interested in, which in turn defines the descriptors used in subscriptions that would populate the routing tables. Since we could not gain access to comprehensive per-user Web-access logs, we instead infer user interests by analyzing the content that users bookmark. We use the bookmark collection of the Delicious website,[2] which contains the public bookmarks of about 950,000 users retrieved between December 2007 and April 2008 [82]. The data set contains about 132 million bookmarks and 420 million tag assignments posted between September 2003 and December 2007. We assume that users are interested in the content they bookmark, and that they describe the content with the tags they assign. Therefore, we derive plausible subscriptions from user tag sets. We slightly clean the data by applying a simple language-based summarization

[2] http://delicious.com

using stemming and removing duplicate tags. In total we derive 123,248,896 subscriptions for 922,651 users.

We also analyze data collected from blogs. In particular, we study the Blog06 collection from the Text Retrieval Conference (TREC),[3] which contains 3,215,171 blog posts from 100,649 unique blogs. We use the latent Dirichlet allocation (LDA) algorithm to extract 400 topics that cover these blog posts. We then assume that an author has an active interest in a specific topic if they write more than two relevant posts on that topic, and consider a post to be relevant only if the probability of the post being classified under that topic is more than 20%. For each topic, we select the 10 most relevant tags and use them as a descriptor of the blogger's interests. Ignoring irrelevant posts and users with no significant interest in any topic, we identify 59,185 blogs with 178,189 relevant posts from which we could derive subscriptions.

Video Content. A future information-centric network will facilitate decentralized distribution of video content. In order to determine which content could be offered by users, which in turn determines the descriptors used in offers, we analyze data from YouTube. Uploaders of YouTube videos can assign keywords to their videos to allow viewers to find those videos with keyword searches. These keywords were publicly visible until three years ago. In particular, we analyze a data set derived from 10,351 videos published by 782 uploaders in the "Politics" category.[4]

Social Messaging. We analyze two different aspects of a Twitter data set to generate workloads for a plausible future messaging service. We take into account the structure of the social graph of followers as well as the content of tweets. We assume that followers are generally interested in the messages posted by the authors they follow. We therefore derive plausible subscriptions issued by the followers. We use a graph of 41.7 million Twitter users and 1.47 billion follower relations. For the content we use a collection of 16 million tweets recorded during two weeks in 2011, corresponding to 1% of the total tweets during that period. This data set was provided again by the TREC conference (2011-2012). A Twitter user can attach a number of "hashtags" to each tweet so that other users can issue searches by hashtag. A user can also include links to other content on the Web. Out of the 16 million tweets, we consider those that have both hashtags and links. We collect the hashtags assigned to each link as a descriptor for that link, and then we use these descriptors as the subscriptions for the users who tweeted that link. In total

[3] http://ir.dcs.gla.ac.uk/test_collections/blog06info.html
[4] http://www.infochimps.com/datasets/11000-youtube-videos

we collect 446,370 subscriptions for 349,753 users.

BitTorrent. BitTorrent constitutes a considerable portion of the traffic of the Internet today. Therefore, it is important to analyze how users describe and access BitTorrent data. We use a data set of 9,669,035 queries collected over a period of 3 months by the Computer Networking Research Laboratory of Colorado State University. This data set contains 1,353,662 unique tags.[5]

General Data Normalization. Some data sets are characterized by large sets of tags (e.g., Delicious). However, when those sets are used to express interests in subscriptions, the specificity of those sets might be excessive, meaning that those descriptors are very unlikely to match any published data. Therefore, in order to normalize those descriptors, we always include at least 5 tags, and then for those descriptors with more tags, we add up to 10 of the remaining tags by selecting them with exponentially decreasing probabilities.

Data Amplification

The extrapolated workloads suffer from two limitations: they are small for the kind of experiments we want to conduct, and they are biased due to the fact that the English language is disproportionately represented in the application traces.

In order to compensate for language bias, but also to expand the size of the workload, we consider other languages that have a meaningful influence on Internet traffic, and we extend the data to include them. We consider the 25 most-spoken languages in the world and amplify all of the data sets according to the distribution of the number of native speakers of those languages.

We do not want to lose the semantic correlations between tags in the data sets, therefore we derive new tags for each artificial language-specific data set by adding a prefix indicating the corresponding language. For example, if the English data set contains the tag "Journalist", we amplify that data set by creating a data set for Japanese where we insert the tag "Japanese_Journalist".

To further expand the workload and also to avoid creating exact replicas of the original data set, we create additional descriptors using synonyms. We assume that at most two tags in every descriptor might be replaced by synonyms, and we choose to replace one, two, or none with equal probability. We also assume that each tag has two synonyms and with equal probability we choose one among them. (We use artificial synonyms.) As an example, the tag "Japanese_Journalist" might be replaced by "sy1_Japanese_Journalist" in some sets.

[5] http://www.cnrl.colostate.edu/Projects/CP2P/BTData/

2.4.4 Memory Requirements

We now evaluate the memory requirements of our ICN routing scheme. For this assessment we develop a synthetic workload corresponding to the plausible behavior of users of different applications over a global-scale information-centric network [22]. We first analyze real traces from four classes of applications: active Web content and blog posts ("push"); video ("pull"); short messages and micro-blogging ("push"); and large BitTorrent downloads ("pull"). We then synthetically expand the resulting workload to 25 other languages that have a meaningful influence on Internet traffic, while preserving the semantic correlation between tags. The full description of these workloads, the methods used to normalize and expand them, and the way we associate users with different applications is detailed in a technical report [67].

Memory Requirements for Inter-AS RIBs

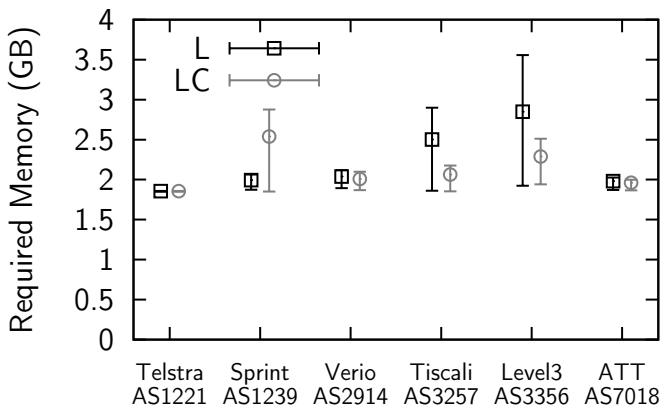

Figure 2.9. Sizes of Inter-AS RIBs.

In Figure 2.9 we show the memory used by the RIBs of the gateway routers of different ASes. This analysis is based on simulations of the routing scheme with 8 trees and under a workload generated for 50 million users. However, since the exact connectivity between the ASes at the level of their gateway routers is not publicly available, we cannot determine how many trees would actually need to be known by each gateway. We therefore simulate all the possible cases and derive the distribution of the memory requirement for every case. The plot shows the minimum, the average, and the maximum amount of memory that would be needed to store the routing information for between 1

and 8 trees. We show the data for the two sets of heuristically derived trees labeled L and LC, as above. The variation is due to the different degree and location of the ASes on different trees. Usually an AS with many neighbors experiences less compression. Notice, however, that the absolute values are relatively low: the most demanding case, which is Level3 with the L heuristic, is less then 3.6GB of memory. Furthermore, the memory required by 8 trees (maximum value), is always less than twice the memory required by a single tree (minimum value). This means that under our scheme, descriptors aggregate well across trees.

Memory Requirements for Intra-AS RIBs

For each AS we also analyze the memory requirements at the intra-AS level. We use the internal AS topologies available from the Rocketfuel project [76]. The data are presented in Figure 2.10. The N and E labels in the graph

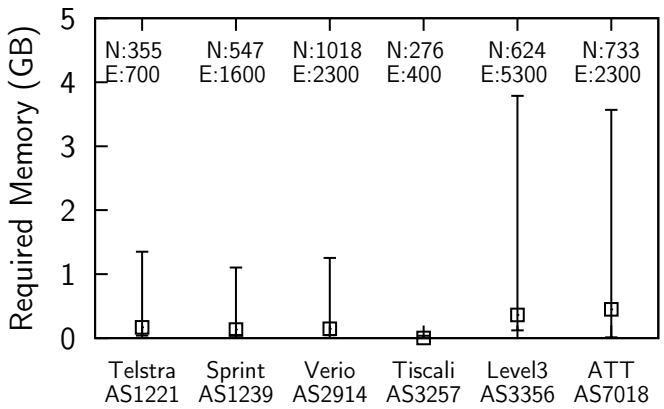

Figure 2.10. Sizes of Intra-AS RIBs.

represent the number of nodes and edges in each AS, respectively. We plot the minimum, average, and maximum sizes of the RIBs used to store local trees. Recall that for the local (intra-AS) trees we store all the shortest-paths rooted at every node (heuristic H3). The number of users inside each AS depends on the distribution of the 50 million users over the AS-level topology. Considering the largest results, namely Level3 and AT&T, we can see that even using a large number of trees (since both have hundreds of routers) we still obtain good levels of aggregation and good results in absolute terms, with a maximum memory requirement of less than 4GB.

Scalability Analysis

The results discussed so far are limited to a relatively low number of users compared to the current population of Internet users. In order to better demonstrate the scalability of our routing scheme, we focus on a particular tier 1 AS (3257) and on a shortest-paths tree derived using heuristic H1 to study the memory requirement under a workload of almost 10 billion content descriptors corresponding to 500 million users. Figure 2.11 shows the memory required

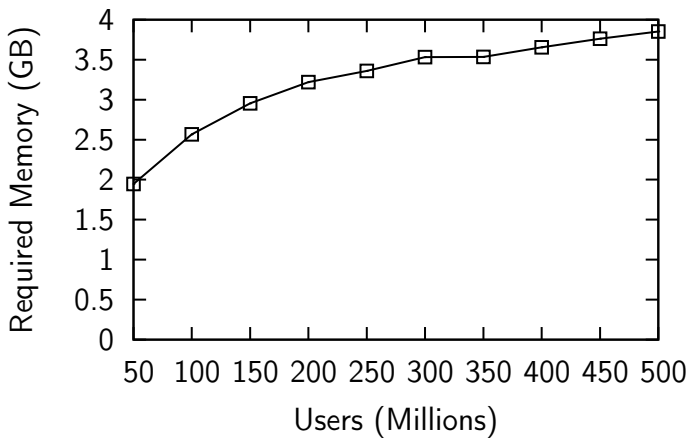

Figure 2.11. RIB Scalability.

for a gateway router for increasingly larger user populations. We can see that the growth of the memory requirement is relatively high initially, but steadily flattens, reaching 3.8GB for 500 million users. This is due directly to the aggregation of tags under our scheme: even with high numbers of users, the memory required to store all the routing information is likely to remain practically constant, since most of the new descriptors will be aggregated at no additional cost.

2.5 Related Work

Although routing is one of the crucial aspects for the development of the notion of an information centric network, there is surprisingly little work on this topic. The NDN project proposes NLSR [40], a link-state routing protocol for NDN. NLSR is a traditional link-state protocol that uses NDN itself to transport routing information. NLSR realizes only a traditional unicast routing scheme that can support multiple paths with multiple runs of the Dijkstra algorithm.

Another interesting work is presented by Papadopoulos et al. [63] who developed two greedy forwarding algorithms in a hyperbolic space. This approach seems promising for routing in ICN and particularly with NDN naming. However, in order to work well in practice, the name space must be hyperbolic, and right now there is no evidence that that is the case. Another problem with this scheme is the relation between the name space and the network topology, meaning how names are distributed over the network. In fact, if names do not follow the same distribution (within the hyperbolic space) then paths can be stretched significantly. Finally, it is not clear how to compute the hyperbolic coordinates of routers and content using only local information.

A number of ICN proposals do not implement a routing scheme based on content names or identifiers, but instead map names or identifiers to network-level addresses. The PURSUIT project[6] uses flat names to identify each object together with a topology/resolution service to obtain a form of network-level unicast or multicast address used for the actual packet switching [44]. DONA [47] also uses flat names and uses a network of special "resolution handlers" to locate the content at the IP network level. NetInf[7] provides a name resolution scheme in combination with a name-based routing scheme similar to CCN/NDN. However, this name-based routing remains localized, so that *global* reachability is only supported through the name resolution scheme [7].

2.6 Summary

We have examined the fundamental problem of routing in an information-centric network, and the essential question of the scalability of routing state. We presented and evaluated a concrete scheme based on trees that supports expressive content descriptors consisting of tag sets. Our evaluation confirms two intuitions: first, that the Internet can be approximated effectively with trees and, second, that tag-based content descriptors, which are more expressive than name prefixes, aggregate well under our scheme.

A crucial problem that needs to be addressed is tag-based forwarding. In the next chapter we present our work on highly parallel forwarding algorithms that combine hardware and software solutions to support high-speed forwarding with tables of hundreds of millions of tag sets.

[6] http://www.fp7-pursuit.eu/
[7] http://www.sail-project.eu/

Chapter 3

TagMatch: A Fast Matching Algorithm for Tag-Based Information-Centric Networking

This chapter presents a fast forwarding engine for the multi-tree tag-based routing scheme we introduced in Chapter 2. At the heart of the tag-based forwarding problem lies the subset matching problem. The significance of the subset matching problem is not only limited to information-centric networking. Subset matching, together with its close relative *superset* matching, often appears in numerous large-scale information processing applications such as publish/subscribe and stream processing systems, database systems, and social media. For instance, in an advanced Twitter-like messaging service where users might follow specific publishers as well as specific topics encoded as tag sets, the service must join a stream of published messages (and their tags) with the users and their preferred tag sets so that the user tag set is a subset of the message tags.

In general within the context of a database system, a subset selection operator selects from a table of sets the subsets of a given query set. This is an old but also notoriously difficult problem that is not amenable to universally effective indexing.

In this chapter we first formalize and contextualize the subset and superset matching problems within the broad area of data processing. We then present *TagMatch,* a system that solves this problem by taking advantage of a hybrid CPU/GPU stream processing architecture. TagMatch targets large-scale data-processing applications with thousands of matching operations per seconds against hundreds of millions of tag sets. TagMatch primary aim is to maximize

throughput.

We evaluate TagMatch on various scenarios that range from message streaming to data mining, with very positive results both in absolute terms and in comparison with existing systems that offer a similar service. As a notable example, our experiments demonstrate that TagMatch running on a single commodity machine with two GPUs can easily sustain the traffic throughput of Twitter even augmented with expressive tag-based selection.

3.1 Subset Matching

While the main purpose of this chapter is to provide a forwarding engine for the multi-tree tag-based routing scheme we introduced in Chapter 2, here we design and explain TagMatch as a general solution for the subset query matching problem. We then discuss the specific use of TagMatch for forwarding in ICN in Section 3.4.5.

Subset matching is essential to global web applications. For example, within the Twitter messaging system, the first stage in ad selection for a user query must find a "match between user attributes and targeting criteria across the corpus of ads,"[1] which in the most basic form amounts to checking that the attributes of the user query contain the targeting criteria of the ads. Even more important for Twitter is the selection of the messages themselves. As of today, Twitter allows users to "follow" a publisher for immediate delivery of published messages. In addition, Twitter provides a keyword or tag-based search over past and current messages. Combining these two features, a user might want to follow a publisher but only on a specific set of keywords or tags, or even just that set of tags without a specific publisher. This data selection based on subset matching is also essential in databases [17, 56, 61], data mining applications [26, 62, 72, 60] where it is used as a selection or join operator, network intrusion detection and virus signature detection [34], packet classification in Internet routers [84], information retrieval systems [73], in some publish-subscribe systems [25] and some information-centric networking architectures [65] where it is used for message brokering and routing.

We start this chapter by giving a formal definition of the subset query problem and several closely related problems. We briefly discuss the state of the art solutions for these problems then we continue by presenting TagMatch as a general subset matching solver. In sections 3.3 and 3.4 we describe the architecture of TagMatch and in Section 3.5 we use TagMatch as tag-based ICN

[1] https://blog.twitter.com/2016/resilient-ad-serving-at-twitter-scale

forwarding engine as well as a modern twitter-like application and evaluate its performance under different realistic heavy workloads. We show that TagMatch achieves very positive results both in absolute terms and in comparison with existing systems that offer a similar service.

3.1.1 Definitions

In Section 2.2.1 we discussed how we use tag sets to represent content descriptors and we explained the matching semantic behind it. The matching problem we are facing in our tag-based forwarding engine is called subset query problem. Basically, given a large corpus of sets $D = s_1, \ldots, s_n$ (the *database*) and a set q from a high-intensity input stream (a *query*), subset matching means finding the sets s_i such that $s_i \subseteq q$. This is a basic combinatorial problem that is also notoriously difficult. Because of the broad usage of subset matching problem in different domains, multiple definitions for the same problem has came to existence. Therefore, in order to study this problem, one has to know about other definitions and closely related problems.

Here we first provide the definition for the *partial matching* problem followed by formal definitions of *subset* and *superset query* problems. We also give the definition for the *set containment* problem that are mainly used in the database literature.

Partial matching problem Given a universe U and a set D of N vectors in $\{0,1\}^{|U|}$, build a data structure, which for any vector query q in $\{0,1,*\}^{|U|}$, detects if there is any $s \in D$ such that q matches s. The symbol "$*$" acts as a "don't care" symbol, i.e., it matches both 0 and 1. Note that one can extend any algorithm solving this problem to handle non-binary symbols in vectors and queries, by replacing them with their binary representations.

Superset query problem Given a set D of N subsets of a universe U, $|U| = m$, build a data structure, which for any query set $q \subset U$ detects if there is any $s \in D$ such that $s \supseteq q$.

Subset query problem The definition of subset query problem is very similar to superset query problem. Here, instead of searching for $s \in D$ such that $s \supseteq q$, we look for $s \in D$ such that $s \subseteq q$. With regard to superset query problem, this problem is essentially the other side of the same coin.

Set containment problem Set containment problem is mostly popular in the database literature [38, 35, 78]. The definition of this problem is as

follows: A database D, where each record t has a set-valued attribute $t.s$ is given. The queries we are interested in are the following:

- *Subset queries.* In subset queries the user asks for all records t that contain the query set q , i.e., $\{t \mid t \in D \wedge q.s \subseteq t.s\}$.
- *Equality queries.* In equality queries the user asks for all records whose set-value is identical to the query set, i.e., $\{t \mid t \in D \wedge q.s \equiv t.s\}$.
- *Superset queries.* In superset queries the user asks for all records whose items are all contained in the query set, i.e., $\{t \mid t \in D \wedge q.s \supseteq t.s\}$.

Already from these definitions, the various problems seem similar if not identical. In fact, all these problems are essentially equivalent. In particular, any instance of subset or superset query can be efficiently transformed into an instance of partial matching, and vice-versa [27, 34], and the set containment problems from the database literature encompass both subset and superset query matching.

However, note that the set containment problems as defined above and in the database literature use a notion of superset query that corresponds to the the definition of *subset* query given above, and vice-versa, a subset query in a database corresponds to a *superset* query (as defined above). This inconsistency of terminology is unfortunate, and it is not even limited to the databases literature. Subset and superset matching problems are often mixed up, perhaps because one can justify the use of either of the definitions depending on how one looks at the relationship between sets. This confusion, together with the duplication of terms, complicates the comparative study of these problems.

To be clear, as described in Section 2.2.1, the original problem we face in our forwarding algorithm is an instance of the *subset query* problem. In particular, the original problem is one in which the size of the universe is not bounded (i.e., no limit on the number of tags). However, in order to make the problem more tractable, we encode each tag set into a 192-bit Bloom filter. With this encoding, we reduce the initial subset query problem (over an infinite universe) to the corresponding partial matching problem over a finite and small universe, $|U| = 192$. Therefore throughout this chapter we assume that tag sets are represented by their corresponding Bloom filters and the subset matching relationship between two tag sets is defined over their corresponding Bloom filters. Note that in our setting, although the size of the universe is relatively small, the number of vectors N is very large.

3.1.2 Existing Solutions and Related Work

It is believed that the partial matching problem suffers from the "curse of dimensionality" [15]. This means that there is no algorithm for this problem which achieves both "fast" query time and "small" space. Hence any solution to this problem faces the trade-off between memory consumption to store the data structure and the query execution time.

A trivial solution would be to store all the N vectors in a single array. To execute a query, a linear scan of all the vectors in needed. Therefore, we need $O(mN)$-bits for the storage and $O(mN)$ time to execute the linear search. Another trivial solution is to store a map of all the possible 2^m query vectors to their corresponding subsets and supersets. Although the time complexity of answering a query is $O(m)$ in this case, the memory required to store this map is in the order of $O(2^m)$. This solution is not practical because the memory required to store this map is exponential in the size of the universe.

The first non-trivial result for this problem has been obtained by Rivest [73]. He proposed two solutions for this problem, one is based on hash-coding and the other one is based on prefix tries. The more interesting of the two is the trie solution. He showed that trie uses a linear storage and achieves sublinear query time. The underlying assumption for his analysis is that the database content is generated at random. Unfortunately in practice, such assumption typically does not hold and the performance of a trie easily degrades if the distribution of the tags in the database is skewed.

There has been other works that try to enhance the trie solution. In Section 2.3.6 we described another solution which uses PATRICIA trie to store Routing Information Base (RIB). PATRICIA trie is a very compact prefix trie. At each node of the trie we store one bit-vector of size m and two pointers to its children, therefore the memory consumption of the trie is $N \times (m + 2 \times |pointer|)$.

Although our PATRICIA trie has a very good performance characteristics, when the size of the trie grows really large, its performance degrades. This happens because in order to execute a query we have to walk over the trie. Since this access is random, we may move data in and out of the cached memory. This is an inherent problem of any tree like data structure. By using a locality-aware prefix trie we can mitigate this problem but the problem emerges again when concurrent threads execute different queries and hence access different parts of the data structure. In [66] we investigated such a locality-aware prefix trie that we designed specifically to solve the subset query problem. The work on the prefix trie is not the main focus of this thesis, hence we avoid going through the details of it and refer you to the publication itself.

Charikar et al. [27] investigated the trade-off between memory consumption and time complexity of any solution for this problem and presented two non-trivial solutions with the following trade-offs:

- $N \cdot 2^{O(m \log^2 m \sqrt{c/\log N})}$ space and $O(N/2^c)$ time for any c
- Nm^c space and $O(mN/c)$ query time, for any $c \leq N$

While the solutions provided by Charikar et al. are valuable from theoretical point of view, unfortunately in practice neither of these two solutions is suitable for our setting. More specifically, note that in the first solution, in order for the trade-off to be beneficial, the number of sets N should not be far smaller than the number of all possible sets in the universe, which is 2^m. We know that in our setting, N is far smaller than 2^m, hence the first solution has low practical value. As for the second solution by Charikar et al., the problem is that the memory requirement is unrealistically high, only to achieve a modest improvement in time over the linear scan.

Roughly speaking, from a practical point of view, there are two kinds of solutions for subset matching. One is to check sets $s_i \in D$ one by one against the query q. Typically, solutions of this kind use "signatures" $sig(p)$ that are compact representations of sets that admit to a fast comparison $sig(s_i) \subseteq sig(q)$ more or less exactly indicative of the relation $s_i \subseteq q$ between the sets [37, 61]. Another solution is to iterate over the elements of the query, $x \in q$, and to use an inverted index to find the list of sets s_i that contain x, and then to combine those lists to find which sets are fully covered by q [42, 59, 11, 8]. This combination of lists can be seen as an exploration of the subsets of the query. In fact, a variant of this second solution looks for the subsets $q_j \subseteq q$ directly in the database (e.g., using a hash table).

Thus both types of algorithms reduce to an iteration over sets and neither one is ideal in all cases: one is a linear scan of the database; the other one iterates over the subsets $q_j \subseteq q$ and therefore is exponential in the size of the query q. Note that trie-based solutions fall into the later category.

Although inverted index is the state of the art approach to answer set containment queries in databases, yet it is shown that their performance degrades when the size of the indexed database becomes very big compared to the domain, or when the distribution of the items is skewed. In these cases, some inverted lists become very long and compromise the performance of query evaluation. The two conditions mentioned above are exactly the conditions we are facing for our fast forwarding algorithm as well as many other real life applications of the subset query problem. Firstly, as we explained in the previous

chapter, our domain size is very small in comparison to the size of the database because we have $(|U| = 192) \ll N$. Secondly, although we use k hash functions to transform tags into k bit positions, still since the popularity of different tags in the original database is not uniform, the frequency of occurrence of these bits is highly skewed as well. This problem with the inverted index solution is investigated in [78].

3.2 Proposed Solution

None of the existing solutions to this problem can achieve the desired performance for our problem size. Also, given the long history of subset matching, we have little hope to achieve a high throughput with a purely algorithmic solution. We therefore take a multi-pronged approach whereby we leverage a hybrid stream processing system of CPUs and GPUs, combining techniques from state-of-the-art subset matching with novel algorithmic and technical improvements on both the CPU and GPU sides. We implement this approach in a subset matching engine called *TagMatch.*

In a very abstract view, TagMatch falls into the category of solutions that use the linear scan of the database to find all subsets of the query. In comparison to other algorithmic approaches, solution that are based on linear scan use much less memory and are less complex. The drawback of this type of solutions is that in order to achieve reasonable speed and throughput, the entire database that can contain huge amount of sets needs to be matched against queries in a relatively short period of time. The task of matching a single query against a single set is relatively a simple. The main challenge is to perform the matching operation against the entire database. Therefore, this approach can benefit from highly parallelized architecture available on graphics processing units. GPU architecture excels in solving problems in which a relatively simple task needs to be perform on large amount of data.

Before introducing TagMatch, we emphasize that the synergistic use of CPUs and GPUs in TagMatch is essential, as demonstrated by the summary results of Table 3.1. Notice first that GPU parallelism alone is insufficient, and in fact it is inferior to state-of-the-art CPU-only solutions, over which TagMatch achieves an almost 10× speedup. Notice further that TagMatch is also significantly faster than the best combination of CPU-only and GPU-only solutions, and that TagMatch itself achieves a 50× speedup when running on a hybrid system. This is because TagMatch exploits the more versatile processing capability of CPUs *in combination with* the massively parallel pro-

	database size		
system	21M(10%)	42M(20%)	212M(100%)
GPU-only, plain	0.40	0.20	0.04
GPU-only, plain with batching	11.50	6.30	1.20
CPU-only, fast prefix tree	21.10	14.00	4.30
CPU-only, state-of-the-art ICN	27.60	17.40	—
CPU-only, TagMatch	3.90	3.40	0.68
TagMatch	268.80	144.40	35.30

(throughput: thousand queries per second)

Table 3.1. Summary Evaluation: Throughput of TagMatch vs. CPU-Only and GPU-Only Systems.

cessing capabilities of GPUs. TagMatch does that with an appropriate division of labor, specific algorithmic solutions, and an effective coordination between CPUs and GPUs. We further discuss an alternative, GPU-only architecture in Section 3.5.5.

To introduce TagMatch, we note that both types of existing algorithms—iterations over $s_i \in D$ or over $q_j \subseteq q$—can benefit from an index to take shortcuts. In the first case, if s_i is not a subset of q, then the iteration can skip all the supersets of s_i. Similarly, in the second case, if q_j is not in the database, then the iteration can skip all the subsets of q_j. So, both solutions can use an index in which sets are arranged according to their subset relations, and one such effective index is a prefix trie (or tree) like the one proposed originally by Rivest [73], which is in fact used in many subset-matching algorithms [42, 56, 66].

TagMatch takes a similar approach, although it indexes *signatures* rather than sets. In addition, TagMatch organizes the data and processing in a platform-specific way, with a coarse-grained index on the CPU side and a fine-grained selection on the GPU side. Specifically, TagMatch uses Bloom filters as set signatures, and partitions the database to divide and coordinate the work between CPUs and GPUs. Appropriately selected bit masks define the partitions and make up the CPU index, which is designed as a compact data structure to support a sequential but memory-efficient matching on CPUs. Correspondingly, signatures are grouped and sorted within partitions on GPUs to enable parallel processing and additional shortcuts. TagMatch then processes queries through a pipeline that alternates CPU and GPU stages. We design and engineer this pipeline so as to maximize the parallelism both between and within each stage.

We validate this design with an extensive experimental evaluation. We measure the performance of TagMatch in several relevant application scenarios both comparatively and in absolute terms. TagMatch outperforms all comparable systems we tested, namely a widely used database system (MongoDB), an existing message forwarding system, and a tightly optimized matcher based on a prefix tree that itself outperforms all subset matching algorithms we know of as reported in the literature. In absolute terms, as a highlight of our results, under a realistic Twitter workload with more than 212 million unique sets representing user preferences, TagMatch can process over 30,000 subset queries per second on a single, commodity machine with two GPUs. And under a realistic ICN workload, it achieves throughput of 383,500 subset queries per second using only one GPU.

Finally, it is important to mention that the performance of our proposed system in the worst case scenario is similar to that of the linear scan solution. Or in other words, its worst case performance is as bad as any other solution for the subset query problem. But in cases where the input data set meets certain condition, it outperforms other existing solutions. TagMatch is designed and optimized to achieve good performance in situations where the size of the universe is much smaller than the number of sets in the database.

3.3 System Model

We design TagMatch as a general-purpose subset matching engine. To make this notion a bit more concrete, we consider sets of string *tags*, which is the most common use in applications and in any case provides a very general model. In essence, TagMatch implements a database of tag sets with a subset-match operation and the interface shown in Table 3.2.

$\textit{add-set}(\textit{set}, \textit{key}) : \textit{void}$
$\textit{remove-set}(\textit{set}, \textit{key}) : \textit{void}$
$\textit{consolidate}() : \textit{void}$
$\textit{match}(\textit{query-set}) : \textit{multiset of keys}$
$\textit{match-unique}(\textit{query-set}) : \textit{set of keys}$

Table 3.2. TagMatch Interface.

The *add-set* and *remove-set* functions add and remove a set with an associated key, where the key is simply a link to application data. These changes

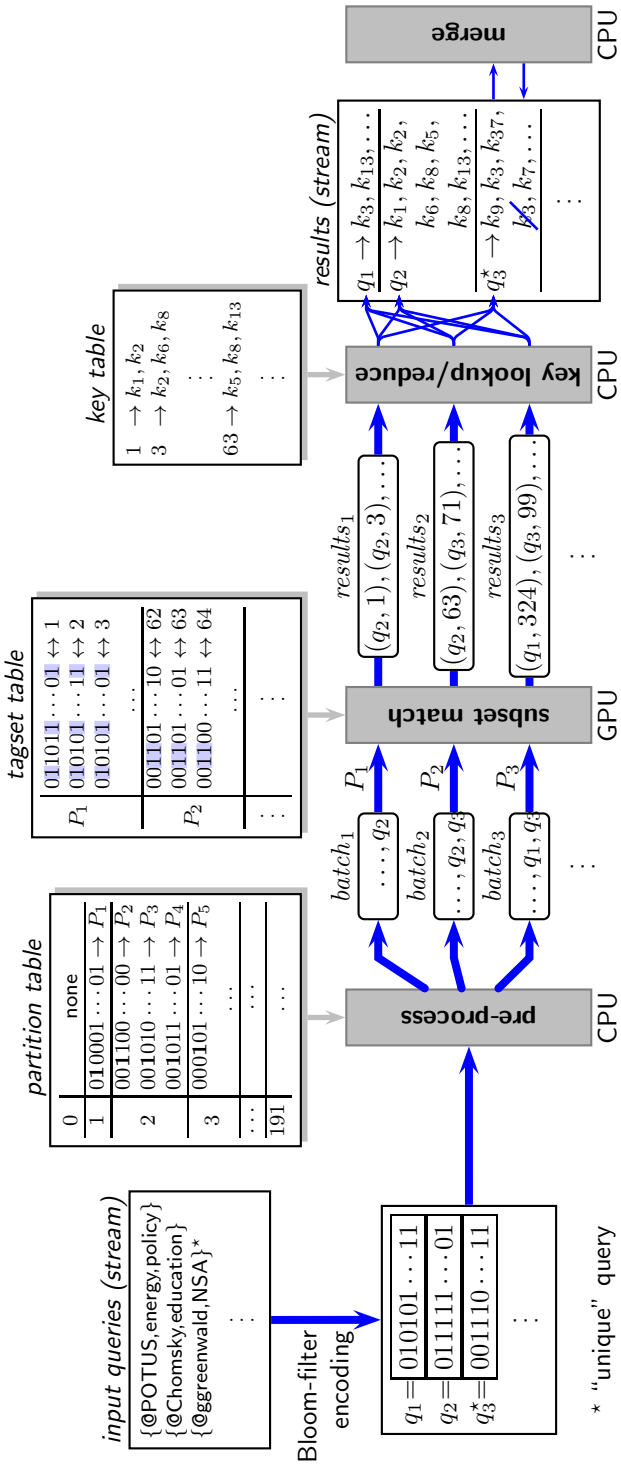

Figure 3.1. The Architecture of TagMatch.

are not immediately effective and instead are staged in a temporary index and become effective only after a call to the *consolidate* function.

The *match* and *match-unique* find subsets of a given query set in the table. *match*(q) returns all the keys k associated with the indexed sets s such that $s \subseteq q$, possibly with multiple instances of the same key k if k is associated with multiple subsets of q. *match-unique* returns a *set* of keys k, such that at least one indexed set $s \subseteq q$ is associated with k (i.e., it avoids duplicate keys).

The *match-unique*(q) function is useful to implement the Twitter-like application discussed in Section 3.1 as well as an ICN router in which each packet has to be forwarded to all the appropriate interfaces but only once per interface. In the Twitter example, the application could store the preferences of users in a table *Users* with two fields: *Users.prefs* and *Users.id*. For each tweet in a stream *Tweets*, the application must find the ids of all the users interested in that tweet, effectively computing an inner join on $Users.prefs \subseteq Tweets.keywords$. Thus the application would use TagMatch to add each user preference u with *add-set*($u.prefs, u.id$), and then to find the matching users for each tweet t with *match-unique*($t.keywords$).

3.4 System Implementation

This section presents the general design of TagMatch, as well as the implementation details of its components. Figure 3.1 shows the high-level architecture of TagMatch, where the gray boxes represent the main computational steps of the *match* and *match-unique* functions while the white boxes represent the main data structures. TagMatch is built on a hybrid CPU/GPU system with one or more CPUs and one or more GPUs. Therefore, some computational steps run on CPUs while others run on GPUs.

At a high-level, TagMatch indexes partitions of related tag sets, and therefore finds the subset of an input query set in two steps: The first step finds the relevant partitions for a query set, and the second step matches the query against the individual sets within those relevant partitions.

Specifically, the matching algorithm consists of a four-stage pipeline: (i) The *pre-process* stage selects the relevant partitions for a query. (ii) The *subset match* stage finds the tag sets that match a query within each partition using a GPU. To maximize throughput and better use the processing capabilities of GPUs, this stage operates on batches of queries, evaluating them in parallel. (iii) The *key lookup/reduce* stage extracts the keys associated with each set of tags, and groups the results by query. (iv) Finally, the *merge* stage combines

the results from multiple partitions into a single set of keys (for *match-unique*) or a multiset (for *match*).

In TagMatch, we do not put any restriction on the universe of tags and it can potentially include any sequence of characters. This brings forth the need for a compact representation for tags and sets. Particularly such representation should support simple ways to perform the subset operation on sets. Because of this, TagMatch represents sets (database and query) as Bloom filters. Bloom filters are an ideal basis for subset matching, since they are compact, fixed-width bit vectors that admit to very simple membership and subset checks. However, those checks are only probabilistically correct and may result in false positives. For sets S_1 and S_2 represented with Bloom filters (bit vectors) B_1 and B_2, $S_1 \subseteq S_2$ implies $B_1 \subseteq B_2$ (bitwise), and $B_1 \subseteq B_2$ (bitwise) implies $S_1 \subseteq S_2$ with high probability, although $S_1 \nsubseteq S_2$ is possible (false positive).

In cases where false positives are absolutely unacceptable, the system or the application can perform an additional exact subset check. In the following, whenever we mention query or database sets, we refer to their representations as Bloom filters, and we implicitly refer to their bitwise inclusion relations.

The width and the number of hash functions that define the Bloom filter representation also determine its false positive probability, and therefore are high-level design parameters that can be optimized for various application domains.

In order to have reasonably small size Bloom filters with reasonably small false positive rate, we have to put an upper bound on the number of tags each set can hold. Based on our observation of different real world applications, the specific implementation of TagMatch that we describe in this thesis assumes that we have at most 15 tags per query or database tag sets. Derived from this assumption we set the size of the Bloom filter in TagMatch to be 192 bits with 7 hash functions. This provides very conservative bounds for false positives in all the application domains we considered.[2]

From the implementation point of view, we know that in order to answer if a query vector q is a superset of the vector p, since both p and q have the same size, we only need to evaluate "(p & ~q) == 0" statement. Note that &

[2]Given two sets $S_1 \nsubseteq S_2$, an m-bit Bloom-filter encoding with k hash functions would result in $B_1 \subseteq B_2$ (a false positive) with probability $P(B_1 \subseteq B_2) = (1 - e^{-k|S_2|/m})^{k|S_1 \setminus S_2|}$, where $|S_1 \setminus S_2| > 0$ is the number of elements of S_1 that are not in S_2. In our case ($m = 192, k = 7$), with a set S_2 of $|S_2| = 10$ tags and another set S_1 that differs by $|S_1 \setminus S_2| = 3$ tags, the Bloom-filter encoding would indicate a false positive with probability 10^{-11}. Roughly the same 10^{-11} false-positive probability exists for a set S_2 of 5 tags and a set S_1 that differs by $|S_1 \setminus S_2| = 2$ tags.

and == are respectively bit-wise logical AND and bit-wise equality operators that operate on bit-vectors. If the result of the evaluation is true, then $q \supseteq p$, otherwise $q \not\supseteq p$.

A vector of size 192 can be represented by three 64-bit blocks, therefore we need to evaluate "(p & ~q) == 0" only 3 times. This subset matching against a single set can be performed very fast and needs only a handful of operations. Hence, using Bloom filters in this way can drastically increase the performance of the linear scan solution.

Note that with this transformation of sets to Bloom filters, the linear scan solutions seems more promising than before. In addition to this, linear scan solution is very memory efficient and is also very cache friendly. It is also easy to parallelize and it remains cache friendly even then. Because of these positive characteristics of the linear scan solution, we decided to utilize this in the design of the TagMatch. The main drawback of the linear scan is that its time complexity is linear, therefore in real world applications where we have a large number of vectors to check, the performance is not acceptable.

One way to improve the performance of linear scan is to limit the scope of the search. To achieve this, we split the set D into smaller partitions and look only into partitions where we have a chance to find a match. This partitioning can be done offline as a pre-processing stage. We will discuss various partitioning approaches in Section 3.4.1.

Although partitioning is very effective in reducing the scope of the problem, in practice we observe that it is not enough to bring down the running time of the system to a desirable level. The main reason is that in each partition, we still have to go linearly to find all the matching sets. This brings forth the main algorithmic challenge of TagMatch which is to perform the subset matching for each partition on the graphics processing unit and utilize the massive parallelization power of GPUs to solve this problem in a reasonable time.

TagMatch stores its index in three main tables in CPU or GPU memory (see Figure 3.1): the *partition table* (CPU) associates each partition with its defining bit mask; the *tagset table* (GPU) associates each tag set s in each partition with a unique id that points to an entry in the *key table* (CPU) that therefore associates tag sets with keys.

TagMatch is designed to exploit parallelism on both CPUs and GPUs. Thus TagMatch can assign any number of threads to the various stages in the processing pipeline. TagMatch may also replicate the tagset table on all available GPUs to match queries in parallel on multiple GPUs. Alternatively, TagMatch can also partially replicate or simply partition an extremely large tagset table

on multiple GPUs.

TagMatch batches queries and results between some processing stages to amortize the cost of transferring information and control between CPUs and GPUs. However, batching may also introduce excessive latency when, depending on the application, some partitions would see a few matching queries over a significant period of time. In those cases, some batches would not fill up and therefore would hold back the queries contained in them. To limit this holding time, TagMatch uses a configurable timeout period after which it automatically processes batches even if they are not full.

We now detail the off-line partitioning of the database and then the on-line processing stages of the TagMatch pipeline.

3.4.1 Off-Line Partitioning

TagMatch indexes the database D in a number of partitions so that all tag sets in a partition share a chosen bit mask (in their bit-vector representation).

Given a configuration parameter MAX_P, TagMatch computes a set of masks that define a set of partitions, each containing up to MAX_P tag sets. More specifically, to make the matching process more efficient, TagMatch computes *balanced* partitions using a recursive partitioning scheme implemented in Algorithm 2. This is done off-line within the *consolidate*() function.

In this algorithm, the way we select *pivot* defines our partitioning strategy and plays an important role in efficiency of the algorithm. Here we briefly discuss three different strategy for selecting the *pivot*.

Most balanced bit. In this strategy, before selecting the pivot, for each unused bit position, we find the frequency of that position being set to 1 in all sets of the current partition P. Then we find the bit position that splits the partition as evenly as possible into two smaller partitions and select it as pivot. We keep applying the same procedure to every newly formed partition that is larger than a pre-configured, maximum allowed capacity MAX_P. Most balanced bit strategy assumes that the presence of each 1-bit in the query—that is also used to form some partitions—, has the potential to filter out some sizable partitions.

Most frequent bit. As the name suggest, in this strategy, after we calculate the frequency for each unused bit position, we pick the bit with the highest frequency as pivot. The thought behind this strategy is that if such a bit is not present in the query, then there is a potential to filter out large partitions.

Algorithm 2: Common-bit Partitioning Algorithm.

Input: database sets D, max size MAX_P
Output: partition table $PT : Mask \rightarrow Partition$
  $PT \leftarrow \emptyset$
  $Q \leftarrow \{((mask = \emptyset) \rightarrow (P = D), (used\_bits = \emptyset))\}$
  while Q is not empty **do**
    extract $(mask \rightarrow P, used\_bits)$ from Q
    if $|P| \leq MAX_P$ and $mask \neq \emptyset$ **then**
      $PT \leftarrow PT \cup (mask \rightarrow P)$
    else
      $pivot \leftarrow$ bit $\notin used\_bits$ chosen according to some heuristic strategy
      $\{pivot$ is a previously unused mask bit that splits P according to some strategy into two parts $P0$ and $P1\}$
      $P_0 \leftarrow \{B \in P | B[pivot] = 0\}$
      $P_1 \leftarrow \{B \in P | B[pivot] = 1\}$
      $used\_bits \leftarrow used\_bits \cup \{pivot\}$
      add $(mask \rightarrow P_0, used\_bits)$ to Q
      add $((mask \cup \{pivot\}) \rightarrow P_1, used\_bits)$ to Q
    end if
  end while

Variable length common prefix. This strategy is one of our initial approaches to tackle the partitioning problem. It has less practical value than the other two and we do not evaluate it in this thesis. In this strategy, unlike the other two strategies that we discussed above, we do not calculate bit frequencies of the current partition. We simply select the next unused bit position as pivot (assuming we start from bit position 0). The advantage of using prefixes as the common feature is that we can simply remove the common prefix bits from each partition, and hence reduce the space needed to store these partitions. Doing so can lead to a better overall performance of TagMatch. But a drawback of this approach is that we only consider certain number of consecutive bits of both queries and partitions to filter out non-matching queries and those bits might not be the most effective bits that can filter out queries.

Notice that for both *most balanced bit* and *most frequent bit* it is also possible to extract the common bits information out of all the vectors of each partition, to reduce the size of the partition. This process can be done offline

for each partition, but if we do so, we need to apply the same modification to every query that we would like to match against that partition. This online bit modification of queries is very expensive and can have a significant negative impact on the matching speed of the TagMatch. Hence, it is better to avoid the compression and fully store each partition.

Also notice that the MAX_P parameter (maximum partition size) can be used to balance the workload between the main processing stages in the TagMatch pipeline. Having a few large partitions would simplify the pre-processing on the CPU side but might overload the subset match on the GPU side. Conversely, small and therefore numerous partitions would reduce the cost of the subset match, but would also increase the cost of the pre-processing. We further discuss and evaluate this trade-off in Section 3.5.

3.4.2 Pre-Process

Given a query set q, the task of the pre-process stage is to forward q for further processing within all the partitions P_i that may contain matching tag sets for q (i.e., subsets of q). Since each partition P_i contains tag sets that share the same bits in $mask_i$, then the task of the pre-process stage is to find all $mask_i$ such that $mask_i$ is itself a subset of q (bitwise).

The pre-process stage uses a simple index for masks that is a kind of inverted index of bit positions (*partition table* in Figure 3.1). Concretely, the partition table is an array PT of 192 vectors of bit masks and the corresponding partition identifiers, where vector $PT[j]$ contains all the bit masks whose leftmost one-bit (bit set to 1) is at position j.

The pre-processing (Algorithm 3) then scans the one-bits of the query set q to classify q. The algorithm is not very sophisticated and yet it is quite efficient in practice because the partition table is very compact and therefore cache-efficient. Also, the concrete implementation of TagMatch uses bit vectors made of 64-bit blocks, so the subset checks in Algorithm 3 ($mask_i \subseteq q$) amount to three simple block operations.[3] It can also be shown that the efficiency of Algorithm 3 does not depend on the distribution of masks over the 192 bits in the partition table.

Whenever the pre-process stage fills up a batch for a given partition P_i, TagMatch extracts all the queries from the queue, copies them to the GPU memory, and invokes the *subset match* kernel for that batch of queries on partition P_i.

[3] In C, `((~q[k] & mask`$_i$`[k]) == 0)`, for block k.

Algorithm 3: Pre-Process Stage.

```
Input: partition table PT, query q
Output: forward q for processing within relevant partitions
  for j ∈ all one-bit positions of q do
    for (mask_i → P_i) ∈ PT[j] do
      if mask_i ⊆ q then
        enqueue q for processing within partition P_i
      end if
    end for
  end for
```

Algorithm 4: Pre-Process Stage in More Details.

```
Input: partition table PT , queue of incoming queries query_queue
Output: forward q for processing within relevant partitions
  while state ≠ shutdown do
    q ← query_queue.dequeue() {this is a blocking operation}
    for j ∈ all one-bit positions of q do
      for (mask_i → P_i) ∈ PT[j] do
        if mask_i ⊆ q then
          batch[P_i].add(q)
          if batch[P_i].is_full() then
            h ← GPU_handler.get()
            output ← subset_match.match(batch[P_i], h)
            process_output(output)
          end if
        end if
      end for
    end for
  end while
```

Algorithm 4 shows the pre-process algorithm with more implementation details. To achieve higher performance, the TagMatch uses multiple CPU threads to run the pre-process algorithm. A single thread is responsible to read the incoming queries and add them to the *query_queue*. This *query_queue* is a single producer multiple consumer queue that is shared among all threads. Every consumer thread runs the Algorithm 4 in parallel to all other threads. Each thread picks a query q from the *query_queue* and runs the pre-process algorithm to find appropriate partitions for it. Note that here Algorithm 3 is

embedded in the Algorithm 4.

Whenever the pre-process algorithm fills up a batch for a given partition P_i, TagMatch extracts all the queries from the batch, copies them to the GPU memory, and invokes the subset match kernel for that batch of queries on the partition P_i. The same thread awaits until the output of subset match becomes available. And then it starts the final processing stage.

For the case of brevity, some details are omitted from Algorithm 4. For example when the batch of the partition P_i becomes full and is sent to the GPU for further processing, any other threads who wants to add a query to the batch of this partition, picks a new batch from a pool of available batches. Whenever a thread finishes processing of the output of a GPU, it frees up the batch and returns it to the pool.

In order to submit a batch of queries to a GPU, several different tasks must be perform beforehand. These tasks are defined in *GPU_handler*. The first task is to copy the queries to a special buffers that are optimized for transferring data between CPU and GPU memory. The memory allocation on the CPU side is pageable by default, but GPU can not access pageable memory directly. Therefore *GPU_handler* uses a special memory allocator to allocate *page-locked* or *pinned* memory. The memory allocated in this way is directly readable by GPUs. This improves the data transfer efficiency between CPUs and GPUs. In addition to this, *GPU_handler* is responsible for calling the GPU matching kernel to perform the *subset match* on the given partition and gets the results back from it. Finally note that in Algorithm 4, all access to query queue and partition queues are synchronized.

3.4.3 Subset Match

The *subset match* stage takes a batch of queries and a single partition of the tagset table, and returns the identifiers of the tag sets that match each query q in the batch.

We develop the subset match on a GPU following the Single Program Multiple Data (SPMD) model using the CUDA framework. With SPMD, one writes a "kernel" function designed to run on a single data item, then invokes that kernel on a set of data items, and the GPU schedules the execution of the kernel on all data items with as many parallel threads as its hardware resources allow.

At a high-level, the subset match kernel processes a single indexed tag set against a batch of queries (see Algorithm 5). The specific tag set is identified

Algorithm 5: High-level Subset Match Kernel.

Input: batch of queries Q, table of tag sets P (partition)
Output: vector of pairs (query,set-id) *results*
{kernel code invoked on each individual entry of partition
$P = (s_1, id_1), (s_2, id_2), \ldots, (s_n, id_n)$; automatic variable *thread_id* identifies the entry assigned to this thread.}
$s \leftarrow P[thread\_id].set$
$id \leftarrow P[thread\_id].id$
for $q \in Q$ **do**
  if $s \subseteq q$ **then**
    atomically append (q, id) to *results*
  end if
end for

by an automatic *thread_id* variable.[4] Notice that, here too, the subset check amounts to a simple operation on each Bloom-filter block. Notice also that the output vector (*results*) is shared by all the threads of a kernel invocation, therefore the append operation uses an atomic increment on the size of the output vector.

Subset Match Optimizations

On the basis of the high-level design of Algorithm 5, we develop and implement several optimizations and performance improvements.

The first and most significant optimization is a pre-filtering step that takes place before the actual subset check. In CUDA, the threads in a kernel invocation are organized in *blocks*, such that all threads within a block run with consecutive thread ids on the same processor and can access a fast (but limited) block-level shared memory.

The pre-filtering exploits the thread-block shared memory as shown in Algorithm 6. The first thread in the block computes the longest common prefix for all the tag sets assigned to the threads in the block, which requires only a simple bit-wise operation between the first and last tag sets in the block thanks to the fact that we store the sets in the tagset table in lexicographical order. Then, all threads in the block iterate through the original batch of queries (in

[4]The CUDA framework defines multiple variables to identify each thread and to control its behavior. For ease of exposition, we abstract from these implementation details and simply refer to a single *thread_id* variable.

Algorithm 6: Pre-Filtering in Subset Match Kernel.

Input: original queries Q', table of tag sets P (partition)
Output: batch of queries Q in shared memory

```
if thread_id = thread_block_first_id then
    first ← P[thread_id].set
    last ← P[thread_id + thread_block_size].set
    len ← leftmost_nonzero_bit(first ⊕ last)
    shared prefix ← first with all bit pos. ≥ len cleared
    shared Q ← ∅
end if
i ← thread_id − thread_block_first_id
while i ≤ |Q'| do
    if prefix ⊆ Q'[i] then
        atomically append Q'[i] to Q
    end if
    i ← i + thread_block_size
end while
```

parallel) to exclude the queries that do not match the common prefix.

A second optimization affects the format of the output of the GPU kernel that needs to be copied to the CPU memory. Copying data from a GPU to the host memory is expensive because the bandwidth of the PCI-Express bus is limited and also because each call to the CUDA API has a fixed, non negligible cost. Therefore, one way to improve performance is to reduce the size of the output of the GPU kernel and to store that output in a single memory region, so as to minimize the number of copy operations.

The output consists of pairs (q, s) for a query q and a matching set s. In practice, we use 8-bit integers to identify a query within its batch, and a 32-bit integer to identify a tag set in the tagset table. However, because of alignment requirements, a simple structure to represent the (q, s) pair would require 64 bits, so a vector of pairs would result in a significant waste (38%) of memory and bus bandwidth. One way to avoid this waste is to store the query and set identifiers in two separate arrays. However, that would require two copy operations. We solve this problem by storing the output vector in groups of four (q, s) pairs, with four packed query identifiers preceding four packed set identifiers:

q_1	q_2	q_3	q_4	s_1	s_2	s_3	s_4	...

This layout yields a 100% or near-100% memory utilization, with a worst-case total loss of only three bytes.

Finally, we apply various fine-grained optimizations to the kernel code. For instance, we manually unroll simple loops and we reduce the number of loop iterations required to read the queries in a batch by accessing two queries within each iteration.

Workflow Optimizations

While the *subset match* kernel exploits multiple GPU cores, running one kernel at a time still can not fully utilize a GPU. This inefficiency is due to the round-trip time incurred in the processing of a batch of queries. When a CPU thread fills a batch of queries within the *pre-process* stage, that thread then must invoke the *subset match* kernel on that batch. In particular, the CPU thread must (1) copy the batch of queries from CPU to GPU memory, (2) invoke the *subset match* kernel on that batch of queries and the corresponding partition, and (3) copy the results back from GPU to CPU memory. And running one such sequence at a time leaves a GPU unused during the copy operations.

To overcome this limitation, and also to parallelize individual kernel executions whenever possible, TagMatch uses CUDA *streams* to enable multiple CPU threads to submit tasks to a GPU concurrently. A stream is an abstraction of a queue of GPU operations. Operations within the same stream execute sequentially in FIFO order, while operations in different streams are executed in parallel as much as possible, depending on the available hardware resources. In TagMatch, each CPU thread that needs to invoke a kernel on a batch of queries acquires an available stream and then issues the sequence of commands for parameter copy, kernel invocation, and result copy, through that stream. The *GPU_handler* mentioned in Algorithm 4 is responsible to perform these operations.

Notice, however, that streams alone do not solve all synchronization problems. Consider the two copy operations. It is immediately possible for an invoking thread to issue a command to copy the minimal amount of data to transfer the batch of queries from CPU to a GPU, because the size of the batch is known at the time of the invocation. However, the same thread can not know at that same time the size of the result. A straightforward solution would be to issue a command to transfer the size of the result, and then only later, when that information becomes available, issue the command to retrieve the results with a minimal transfer. However, this would introduce an additional round-trip time and an additional synchronization point.

In TagMatch we avoid this inefficiency by associating each GPU stream with *two* buffers for the results (call them *even* and *odd*), each containing a length and a set of results. We then alternate between the two buffers as follows. In an odd transfer cycle, we use the odd buffer to transfer the length of the next (even) set of results, as well as the set of results for the current (odd) cycle. And for this copy operation we can issue a command with minimal transfer size because the exact size of this (odd) set was transferred in the previous (even) cycle and is readable from the even buffer. Then, similarly in the following even cycle, we find the length of the current (even) set of results in the odd buffer, which we use to issue the copy command for the current (even) set of results, and so on.

In summary, TagMatch takes full advantage of streams, with the following key benefits for the pipeline architecture. First, GPUs can process multiple batches on multiple partitions in parallel. Second, communication between CPUs and GPUs achieves an optimal utilization of the bus in both directions. Third, CPU threads can invoke entire sequences of GPU operations asynchronously, which means that CPU threads are no longer responsible for the synchronization between copy and processing operations, which in turn allows them to continue with pre-processing, key lookup/reduce, and merge tasks. Finally, TagMatch splits the workload across all available GPUs, with maximal inter-GPU parallelism in the case of full replication of the tagset table.

3.4.4 Key Lookup/Reduce and Merge

As shown in the previous section, the *subset match* stage outputs results in the form of (q, s) pairs, where q is a query id and s is a unique identifier of a tag set in the tagset table.

When new results become available for a partition, a CPU thread picks up these results and performs the *key lookup/reduce* stage, which accesses the *key table* to retrieve the set of keys associated with each set-id s. The thread then groups these keys by query in a *results table* (see Figure 3.1), associating each query with a list of sets of keys. Additions to the *results table* also use the proper atomic operations to allow access from multiple threads.

For each query q going through the matching pipeline, TagMatch maintains a counter of all the batches (partitions) within which q is forwarded for processing. When q's pre-processing terminates, and the counter goes back to zero, signaling that all the results for all the batches returning from the GPUs have been accounted for, then TagMatch runs q through the last *merge* stage. In the case of a *match* query, that requires no additional processing. In the

case of a *match-unique* query, the merge stage merges all sets of keys associated with q into a single set.

3.4.5 TagMatch Adaptation as an ICN Message Forwarder

In Section 3.3, we laid out the system model and architecture of TagMatch as a general subset matching engine. The main reason to develop this system was to use TagMatch as a fast and high throughput forwarding engine to accompany the tag-based routing scheme that we introduced in Chapter 2. To be able to use TagMatch as an ICN message forwarder, we slightly modified TagMatch and introduced the logic of routing trees and interfaces to it. Here we briefly discuss the list of changes we applied to TagMatch.

First, as stated in Section 3.3, each tag-set is associated with a set of *keys*, and we use a key to store a pair *(tree, interface)*. Each tag-set together with its set of keys represents a forwarding rule. Second, we store the table containing the mapping *set-id* $\rightarrow$ *keys* in the GPU memory. Third, each query is an incoming ICN packet that, in addition to its tag set, carries a pair (t, i) indicating that the packet is forwarded along tree t and was received from network interface i. For each query/packet, we supply the (t, i) pair associated with that packet to the GPU backend. Fourth, we modify Algorithm 5 such that, for each query, the algorithm returns a list of interfaces associated with matching entries that are also on the same tree t. The modified algorithm also excludes interface i from the set of output interfaces. This ensures that the packet will not be forwarded back to where it came from. Finally, we modify the *match-unique* function such that for query q it returns all the unique interfaces that q has to be forwarded to. These modifications enables TagMatch to operate as a forwarding engine for our multi-tree tag-based ICN routing algorithm.

3.5 Evaluation

We now present the results of an experimental evaluation of TagMatch. The general objective of this evaluation is to assess the performance of TagMatch in terms of throughput. Most importantly, we are interested in (1) the effective throughput measured in queries processed per time unit under realistic workloads, (2) the scalability of TagMatch with respect to the size of the database and queries as well as to the capabilities of the platform (e.g., available CPU threads), and (3) the performance of TagMatch relative to other comparable state-of-the-art systems.

3.5.1 Subjects and Experimental Setup

The main subject of our experimental analysis is a C++ implementation of TagMatch as described in Section 3.4. In addition, we use the following subjects:

- *prefix tree:* a main-memory implementation of a subset matching algorithm that indexes database sets into a prefix tree. Specifically, this system uses a PATRICIA tree and solves the subset matching problem by navigating that tree. This implementation is representative of most state-of-the-art approaches based on trees (see Section 4.2).

- *ICN matcher:* an implementation of a state-of-the-art algorithm specifically designed to perform packet forwarding in Information Centric Networks (ICN) [66]. In this context, the database encodes forwarding information represented as sets of tags, and the queries are the packets to dispatch. This algorithm is also based on a prefix tree and, similar to TagMatch, it is designed for high throughput.

- *MongoDB:* the *MongoDB* Database Management System (version 3.2.10), which offers an explicit subset operator.

Our testbed is a general-purpose machine equipped with two Intel Xeon E5-2670 v3 processors, each with 12 cores running at a clock frequency of 2.30GHz, and 64GB of RAM. The machine also has two NVidia TITAN X graphic cards each with 12GB of GDDR5 RAM.

To make the comparison as fair as possible, we configure the prefix tree and the ICN matcher to use Bloom filters with the same size as TagMatch (192-bit), and we try to feed the same input and allocate the same system resources to all subject systems. Thus for all the comparative experiments, we give each system the same number of threads.

Still, we could not perform exactly the same experiments with all systems. In particular, since the ICN matcher uses a significant amount of memory to build its index, we could only test the ICN matcher with a reduced portion of the largest workloads. We discuss this case in Section 3.5.3. We encounter analogous and even more extreme difficulties with MongoDB. In fact, the performance of MongoDB is limited to the point of making larger experiments impossible or pointless. We therefore test MongoDB with specially crafted and relatively small workloads. We discuss the case of MongoDB in Section 3.5.4.

3.5.2 Workloads

We evaluate the absolute performance of TagMatch using two independent workloads one representing a forwarding engine of a tag-based ICN router which we refer to as ICN workload and the other one representing a Twitter-like messaging system which we refer to as Twitter workload. The ICN workload consists of a data set of tag-sets that represents the forwarding table of an ICN router and a set of incoming messages (packets) as queries. In the Twitter workload, the database entries are tag sets that represent the interests of users, the keys associated with each tag set are the identifiers of the users interested in that tag set, the queries are the tweets published by the users, and the tags in the queries are the hash-tags (keywords) of the tweets. In the following we describe how we created these two.

Twitter. The Twitter workload includes 300 million users (keys), which is roughly the number of users that are active on Twitter every month,[5] and contains over 212 million unique sets representing user interests.

We generate the set of interests based on a real data set of tweets provided by the TREC conference (2011-2012), containing 16 million tweets recorded during two weeks in 2011.[6] To derive realistic relations between users, we use a graph of 41.7 million Twitter users and 1.47 billion follower relations [52]. To amplify the data set and to prevent a bias toward the English language in the workload generation, we artificially create multiple languages in our data set: given a tag, we "translate" it by adding a prefix that indicates the new language. For example, the original tag *cat* becomes *fr_ cat* in French or *it_ cat* in Italian. In our workload we assume that 40% of the users speak only one language while the remaining 60% speak two languages[7]. To select the language spoken by each user we use two different distributions: the first one is the language distribution on Twitter [39], while the second one is the distribution of the most frequent second languages used in the world[8].

For each user in our workload we generate a set of interests as follows. First we select the languages spoken by the user according to the distributions mentioned above. Then we pick the number of followed publishers according to the follower distribution that we derive from the Twitter

[5] Twitter stats: https://about.twitter.com/company
[6] https://github.com/lintool/twitter-tools/wiki/Tweets2011-Collection
[7] http://ilanguages.org/bilingual.php
[8] https://www.ethnologue.com/statistics/size

graph. Then we randomly select the publishers from the list of users available in our data set and collect their tweets. We generate one interest for each publisher by randomly selecting one of their tweets and using the hash-tags in that tweet. In addition, we "translate" the hash-tags using one of the two languages assigned to the user, since we assume that a user follows only publishers that write in one of the user's languages.

If the publisher of the tweet is a frequent writer, we also add the id of the publisher as a tag in the interest. We consider a publisher to be a frequent writer if he or she is ranked in the top 30% based on the number of published tweets. An interest with only hash-tags describes the set of information that the user wants to collect, while an interest that includes a user id selects only the information of interest that are generated by the user with that id. This procedure results in interests containing an average of five tags.

ICN. In the previous chapter, we created a synthetic but realistic application workload for Information centric network. The ICN data set we use for evaluation of TagMatch engine is the one generated during the experiment reported in Figure 2.11. This workload contains more than 91 million routing entries or in the other words, routing rules. Each routing rule consist of a Bloom filter f, a tree identifier t and an interface i. If an ICN router receives a message q on tree t' and interface i', it should find all the entries f in the routing table such that $q \supseteq f$ and $t' = t$ and $i' \neq i$, then forwards the message q to all corresponding interfaces. This 91 million routing entries is consist of 63 million unique Bloom filters. We create our forwarding table by indexing the entries according to Bloom filters. This allows us to compress the forwarding table as depicted in Figure 3.2. In this compressed from, each entry in the forwarding table points to a set of *(tree, interface)* pairs.

uncompressed forwarding table

0	$100101 \cdots 01 \rightarrow (t_3, i_2)$
1	$011101 \cdots 01 \rightarrow (t_1, i_7)$
2	$011101 \cdots 01 \rightarrow (t_2, i_6)$
3	$001011 \cdots 01 \rightarrow (t_1, i_7)$
...	...
$91m$	...

compressed forwarding table

0	$100101 \cdots 01 \rightarrow \{(t_3, i_2)\}$
1	$011101 \cdots 01 \rightarrow \{(t_1, i_7), (t_2, i_6)\}$
2	$001011 \cdots 01 \rightarrow \{(t_1, i_7)\}$
2	$000100 \cdots 01 \rightarrow \{(t_5, i_9), (t_5, i_{75}), (t_8, i_{134})\}$
...	...
$63m$	...

Figure 3.2. Compression of Forwarding Table.

Unless otherwise stated explicitly, in most of the following experiments we use the Twitter workload with *most balanced bit* as partitioning strategy. In some other experiments we use and explicitly refer to the ICN workload.

Queries

One method to generate queries is to select the tags in each query more or less uniformly at random. These are the hash-tags in each tweet, or tags in ICN interests. However, that would most often result in queries that are immediately and very efficiently discarded in the initial pre-filtering stage. So, to be conservative, we instead create a workload in which each query matches at least one tag set in the database. To do that, we generate each query by selecting a tag set from the database to which we then add between two and four extra tags selected at random. (We also experiment with a broader range of additional tags; see Section 3.5.3.) The rationale for this generation algorithm is that the selected set from the database would perhaps represent a generic topic while the additional tags would characterize the specificity of a tweet or ICN message.

Beyond that, as we said, the intended effect of this method is to obtain conservative results for the matching throughput, since the method essentially forces every query to go through the subset match phase on the GPU stage and then the key lookup/reduce and merge phase on the CPU (in the case of *match-unique*). We apply the same technique to generate ICN queries as well as Twitter queries.

3.5.3 Performance and Scalability

We now present various series of experiments intended to test the performance and scalability of TagMatch under a variety of workloads and configurations. In these experiments we first measure the effect of partitioning parameters and different partitioning strategies on TagMatch. Then we measure the throughput in terms of number of processed queries per seconds, and later we measure the matching latency. We analyze the performance of TagMatch in absolute terms and also in comparison with the state-of-the-art prefix tree.

Size of the Query Set

In the first series of experiments we test the performance of TagMatch and the prefix tree with queries of increasing sizes. Figure 3.3 shows the results of these experiments. As explained in Section 3.5.2, the primary workload

we use consists of queries with between two and four additional tags, which corresponds to the histograms at positions 2–4 in Figure 3.3.

It is clear from the figure that the number of tags in each query has a very significant impact on performance (notice the log scale). This is intuitive from an algorithmic perspective, since query sets of higher cardinality are likely to lead to more one-bits in the Bloom filters, which are likely to match more prefixes and therefore require more data transfer between CPUs and GPUs, and also more work on both sides.

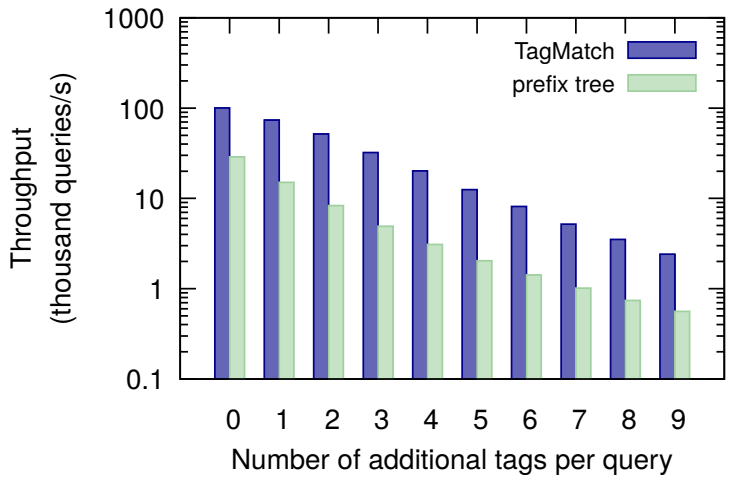

Figure 3.3. Average throughput for *match-unique* with queries of different sizes.

However, notice that for the same intuitive reasons, the decline in *input* throughput, which is what we measure in Figure 3.3, does not result in a corresponding decrease in *output* throughput. In fact, as it turns out, and as we demonstrate with the measurements of Figure 3.4, the output throughput for the same experiments increases significantly with the query size. We argue, intuitively, that for selective queries, meaning queries that have a few matching sets and that represent tweets in the long tail of popularity, the most important performance metric is the input throughput. Conversely, for queries with a high fan-out, which represent highly popular Twitter traffic, the limiting factor and therefore the most interesting performance metric is the *output* throughput. And in this respect, the experiments show that TagMatch performs quite well.

TagMatch is also consistently faster than the prefix tree system by almost one order or magnitude for both the input and output throughput. The results for *match* (not shown) are very close to the results for *match-unique*.

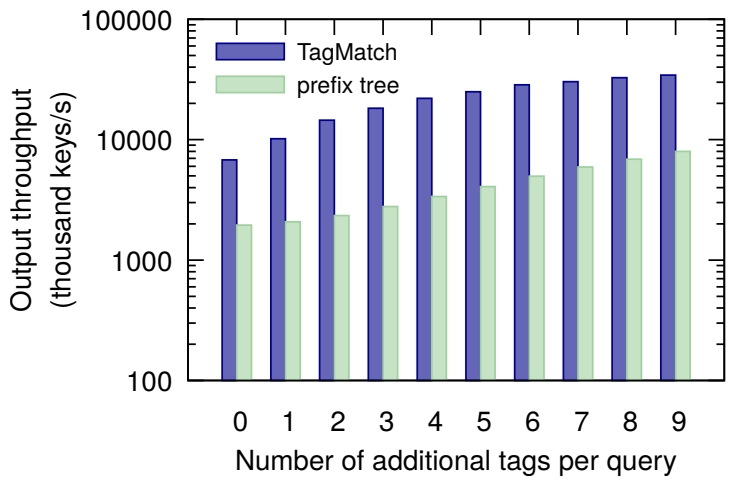

Figure 3.4. Average output rate for *match-unique* with queries of different sizes.

Size of the Database

In a second series of experiments, we test the scalability of TagMatch with respect to the size of the database. We report the results of this analysis in Figure 3.5. Once again we measure the throughput (input) as we vary the size of the database from 20% to 100% of the entire Twitter database of 212 million tag sets.

The salient result of this analysis is that TagMatch can process more than 30 thousand queries per second in the case of *match-unique*, and more than 35 thousand queries per second in the case of *match*, with the full database of 212 million unique sets. This is well above the entire traffic of Twitter, which was on average 6000 tweets per second as of 2015—on a single commodity machine, with the added capability of filtering tweets based on their content. In contrast, the state-of-the art CPU implementation based on a prefix tree can process about 4400 queries per second both in the case of *match* and in the case of *match-unique*.

As Figure 3.5 shows, and as one would also expect intuitively, the size of the database significantly affects performance: with a database with 20% of the entries of the full Twitter workload, TagMatch can achieve a throughput of over 130K queries per second in the case of *match-unique* and more than 140K queries per second in the case of *match*, compared to the CPU implementation that achieves a throughput of less than 14K queries both in the case of *match-unique* and in the case of *match*.

Table 3.3 compares the throughput of TagMatch with the ICN matcher [66].

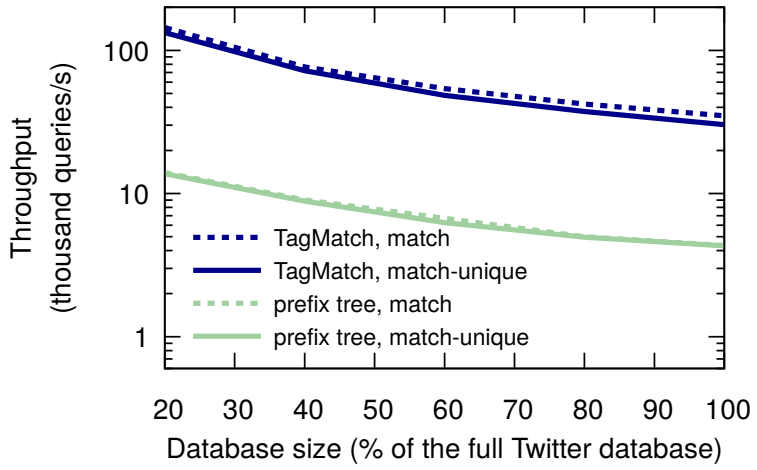

Figure 3.5. Average Throughput for *match* (left) and *match-unique* (right) with Different Database Sizes.

	database size			
	21M(10%)	42M(20%)	21M(10%)	42M(20%)
system	*match*		*match-unique*	
TagMatch	268.8	144.4	249.3	133.0
Prefix tree	21.1	14.0	21.0	13.8
ICN matcher	27.6	17.4	27.5	16.8

(thousand queries per second)

Table 3.3. Comparison with the CPU Prefix Tree, CPU Algorithm for ICN. Average throughput for *match* and *match-unique* with 10% and 20% of the full Twitter database.

In this case we could only consider up to 20% of the full Twitter database because the implementation of the ICN matcher requires a lot of memory during the construction phase to generate the final index that is actually used for the matching. Creating the index for databases larger than 20% of the full workload would require more than the 64GB of main memory available on our machine. The ICN algorithm reaches a higher throughput than the CPU prefix tree algorithm, but remains about an order of magnitude slower than TagMatch.

Effect of the Pre-filtering on GPU Kernel

We design the next set of experiments to evaluate the effectiveness of Algorithm 6. In these experiments we use a simple CUDA kernel that only performs the high-level subset match explained in algorithm 5. As expected, the pre-filtering allows TagMatch to achieve a higher throughput than with the simple kernel. Table 3.4 shows that the throughput of TagMatch is significantly higher for a database size of 21M entries. However, this throughput gain shrinks for larger databases. Specifically, for the 21M database, the pre-filtering improves the throughput by a factor of 2.52. However, on the full Twitter database, the gain is only 1.43.

This shows that the effect of pre-filtering is less significant when we use our largest database. We suspect that the reason behind this is the load on the CPU part of the pipeline. Our investigation reveals that in this scenario, the *key lookup* stage of the TagMatch pipeline is the bottleneck of the system. In order

	database size		
system	21M(10%)	42M(20%)	212M(100%)
TagMatch	268.80	144.40	35.30
TagMatch, simple kernel	106.71	65.86	24.59
TagMatch, simple kernel, no lookup	106.88	66.33	25.58
TagMatch, no lookup	343.64	227.73	75.94

(throughput for *match* queries: thousand queries per second)

Table 3.4. Effect of Kernel Optimizations on TagMatch.

to fully focus on the effect of the GPU kernel, for the next set of experiments, we disable the *key lookup* stage of TagMatch to make sure the CPU is not the performance blocker. In this new setting, we design two experiments, one with the simple kernel and one with pre-filtering. Disabling the *key lookup* stage slightly improves the throughput of the TagMatch with simple kernel, and achieves 25.58 thousand queries per second for the full Twitter workload.

On the other hand, TagMatch without *key lookup* (with pre-filtering) unleashes the power of the pre-filtering kernel and achieves throughput of 75.94 thousands queries per second which is 3× better than the TagMatch without *key lookup* and with simple kernel.

Number of Threads

We now test the ability of TagMatch to distribute its work load over multiple threads. In particular, we measure the throughput as we allocate an increasing number of threads to the CPU stages.

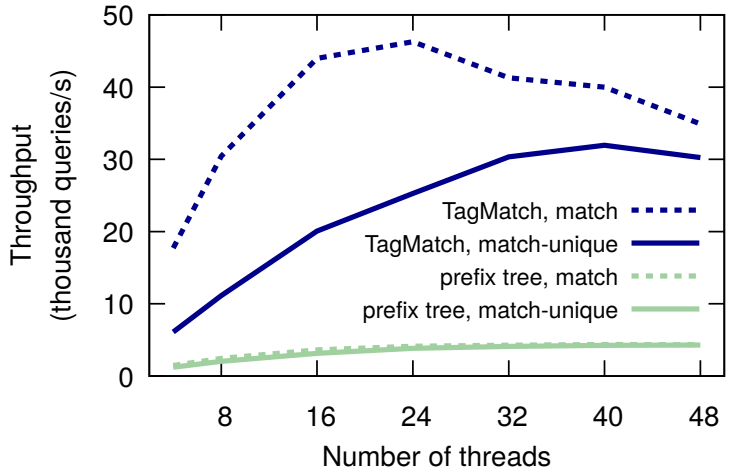

Figure 3.6. Average Throughput for TagMatch and the CPU Prefix Tree with Different Numbers of CPU Threads.

As shown in Figure 3.6, for both *match* and *match-unique* queries, TagMatch achieves an almost linear scalability in the number of threads, with a speedup of more than 1.8× from 4 to 8 threads, and 3.3× from 4 to 16 threads. With more than 24 threads, the throughput for *match* decreases, while the throughput for *match-unique* keeps growing up to more than 40 threads. The difference between the two algorithms is simply due to the higher CPU load of the merge stage for *match-unique*.

The decrease in parallelism speedup over a certain number of threads is instead due to a limitation of the GPU architecture. Basically, beyond a certain number of threads, the GPU stages become the bottleneck for the whole pipeline. However, there are also other factors that limit the pipeline to an overall throughput that is lower than the maximal throughput of the GPU. In particular, CPU threads and the GPUs interact through a set of GPU "streams" (see Section 3.4.3), and on our platform we can allocate a maximum of 20 streams (10 per GPU), primarily due to memory limitations. Having more streams would allow for more parallelism. Furthermore, our test machine has 24 real cores, so when we allocate 32, 40, and 48 threads, those run using Intel's Hyper-Threading technology.

Latency

The batching of queries in the TagMatch pipeline is essential to achieving high throughput but also induces a latency overhead. To limit latency, an application can set a timeout after which a batch of queries will be pushed through the GPU (see Section 3.4). In Figure 3.7 we characterize the distribution of the matching latency for different timeout settings, including the case with no timeout.

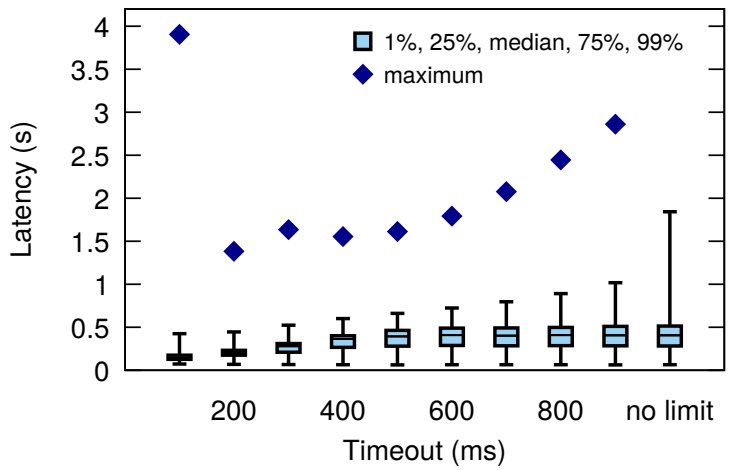

Figure 3.7. Distribution of the End-to-End Latency for *match-unique* with the Full Twitter Database.

These experiments show that the timeout mechanism is indeed effective in reducing latency. Even without a timeout limit, the vast majority of queries (99%) incur a latency of less than 2 seconds, with a median latency of under 400ms. However, the maximal latency values, also shown in Figure 3.7, can be significantly higher.

The case in which the timeout is set to 100ms is particularly interesting. Excluding the case with no timeout limit, the 100ms setting is the one with the highest maximal latency at nearly 4 seconds. This is due to the fact that a very short timeout leads to inefficiencies in the use of the CPU/GPU pipeline. In particular, a short timeout triggers too many invocations of the GPU matching kernels with batches of only a few queries, and since the matching kernel requires the same amount of GPU resources even for small batches, this increases the load on the GPU without a corresponding increase in throughput. In fact, with a timeout setting of 100ms, TagMatch suffers a loss of overall throughput of about 20% (24 thousand *match-unique* queries per second).

However, this inefficiency disappears very quickly with a slightly higher timeout setting. A timeout as short as 200ms already enables TagMatch to process more than 28 thousand queries per second, and a timeout of 300ms further increases the throughput to 30 thousand queries per second, which is close to the maximum achievable with no timeout limit at all.

Balance Between CPU and GPU Load

We now study another algorithmic aspect of the TagMatch pipeline that balances the load between CPUs and GPUs. As discussed in Section 3.4.1, TagMatch uses a configuration parameter MAX_P to define the maximum number of tag sets in each partition. Therefore, for a given set of database sets, MAX_P controls the balance between the number and size of the partitions, which in turn can balance the load between the pre-processing phase (on CPUs) and the subset match phase (on GPUs). Large partitions simplify the pre-processing phase, but might overload the subset match, while several small partitions reduce the complexity of the subset match but increase the cost of pre-processing, as well as the duplication of queries into multiple batches.

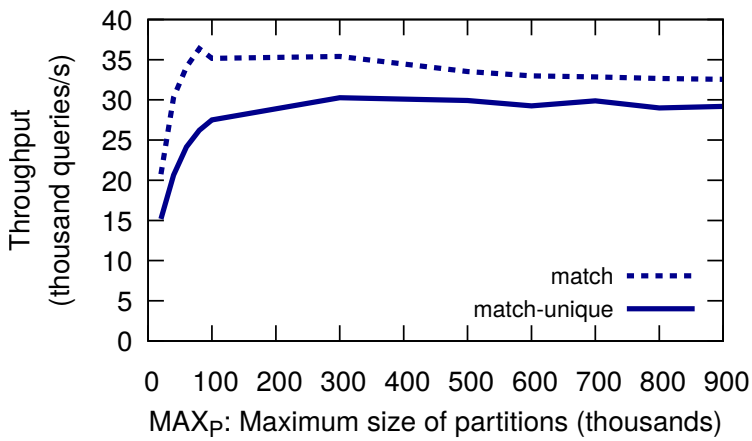

Figure 3.8. Average Throughput of TagMatch for *match* and *match-unique* with Different Size of Partitions.

Figure 3.8 shows the results of an experiment intended to analyze this trade-off. The chart shows how the throughput of TagMatch changes with different values of MAX_P for the same database. We observe that TagMatch achieves the best performance with around 200K tag sets per partition, and

that the throughput remains stable after this threshold. The results do not differ significantly in the cases of *match-unique* and *match*.

It is also important to study the effect of different partitioning strategies on the performance of TagMatch. For the following experiments we use the ICN workload and a single GPU device.

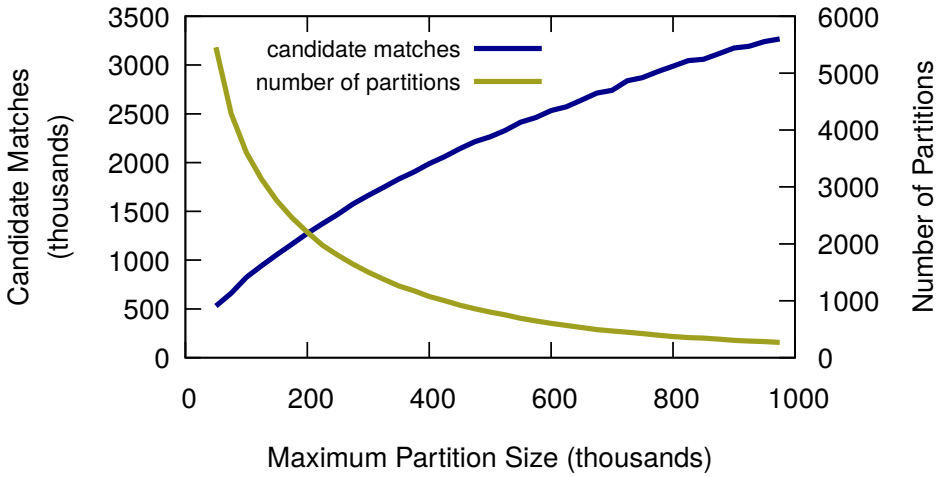

Figure 3.9. Partitioning Algorithm with Most Balanced Bit as Pivot.

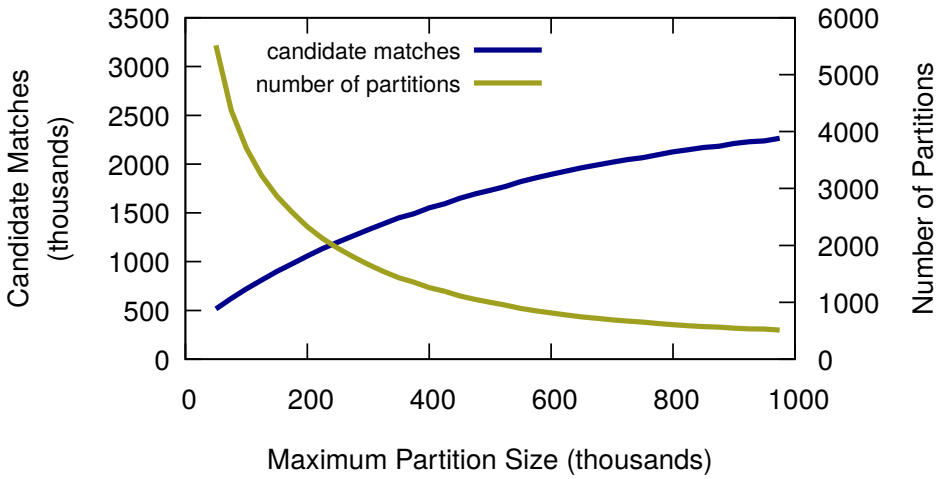

Figure 3.10. Partitioning Algorithm with Most Frequent Bit as Pivot.

In Figure 3.9 and 3.10 we measure the impact of MAX_P on the total number of partitions and the average size of the search space. Notice that these experiments study the impact of MAX_P on the search space independently from the

TagMatch engine. In both figures, we can clearly see that when MAX_P grows, Algorithm 2 produces fewer but bigger partitions. It also shows that smaller partitions limit the average size of the search space much better than large partitions. In these figures, *Candidate Matches* shows the average number of tag sets we need to check per query.

The results also show that the *most frequent bit* produces slightly more partitions than the *most balanced bit* for the ICN workload, but interestingly, it performs better in limiting the search space. In other words, the number of candidate matches that the GPU kernel has to scan (linearly) to find subsets, is lower when we use the *most frequent bit* strategy for partitioning.

As discussed in Section 3.4.1, we propose three different strategies for selecting a *pivot* in Algorithm 2. Each of these strategies directly influence the quality of the resulting partitions. Quality here means increasing the probability that the resulting partitions would *filter out* incoming queries. That is, the probability that none of the sets in each partition would match the query, so that partition would not have to be examined by the GPU back-end. In other words, a good partitioning is one that would limit the overall size of the search space for each query (in expectation).

From the three mentioned strategies, in practice we notice that the improvements we gain from memory compression of the *variable length prefix* strategy is not effective enough, and this method is not able to produce high quality partitions. Hence we exclude the results of this strategy from our evaluation.

In figures 3.11 and 3.12, we measure the throughput of TagMatch for varying MAX_P sizes and different number of CPU threads for both *most balanced bit* and *most frequent bit* strategies. TagMatch with *most balanced bit* strategy achieves its highest of 361,400 queries per second with 6 threads when MAX_P is set to 200K. For *most frequent bit* the highest throughput is 383,500 queries per second again with 6 threads and when MAX_P is set to 225K.

Note that in Figure 3.11, after reaching the highest throughput, increasing MAX_P has a negative impact on the throughput of TagMatch. This effect is less significant for the *most frequent bit* strategy depicted in Figure 3.12. In other words, the performance of TagMatch while using the *most frequent bit* strategy is less susceptible to changes of the MAX_P parameter. One possible explanation for this behavior is that, as evidenced in figures 3.9 and 3.10, the growth in the number of average candidate matches is slower for the *most frequent bit* than the *most balanced bit* strategy. This means that, although for large values of MAX_P we have larger partitions, the *most frequent bit* strategy produces partitions that are better at filtering out the queries.

We emphasize the fact that *both* the partitioning strategy and the MAX_P

value have a significant impact on the performance of TagMatch. Therefore, both parameter have to be tested and optimized for different application domains. For example, for the Twitter workload, TagMatch achieves the highest throughput using the *most balanced bit* strategy and a maximum partition size of 200K.

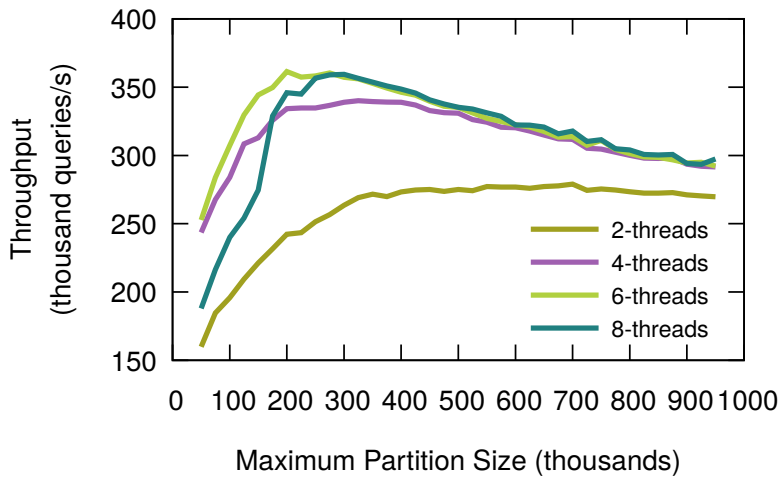

Figure 3.11. Most Balanced Bit Strategy.

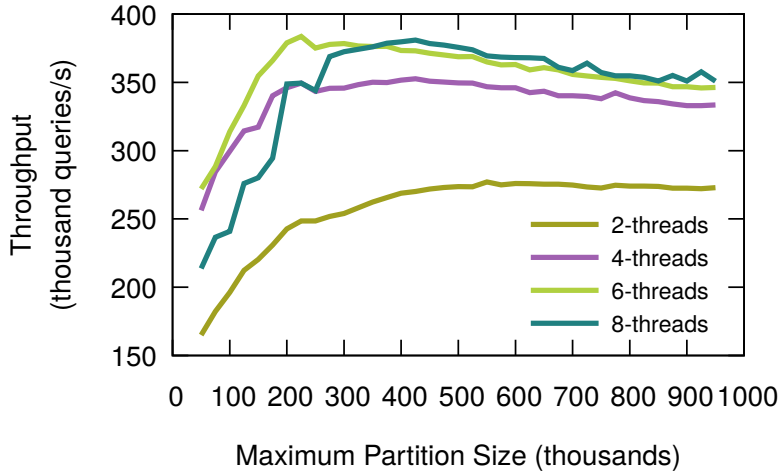

Figure 3.12. Most Frequent Bit Strategy.

Off-Line Partitioning Costs and Memory Usage

TagMatch provides two functions, *add-set* and *remove-set*, to add or remove sets from the database. However, these changes become effective only after a call to the *consolidate* function, in which TagMatch builds its partition and tagset tables using the balanced partitioning of Algorithm 2. We now evaluate the performance of the partitioning algorithm as well as the memory usage on both the CPU and GPU side.

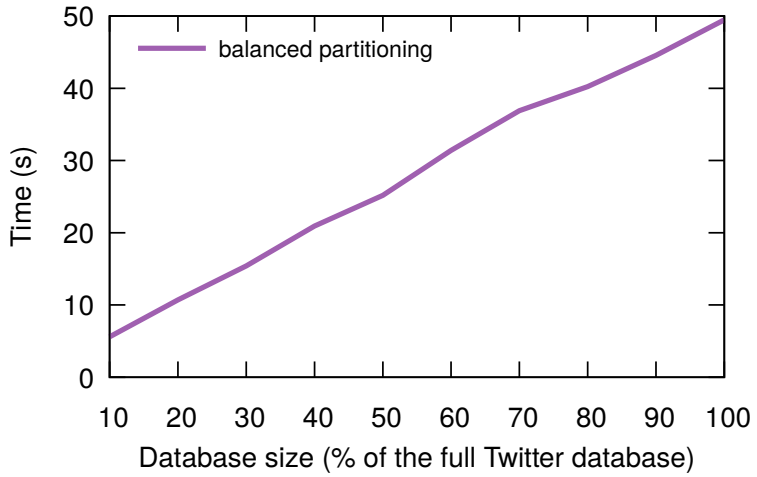

Figure 3.13. TagMatch Partitioning Time, $MAX_P = 200$K.

Figure 3.13 shows the running time of the partitioning algorithm as a function of the size of the database. The experiments confirm that the algorithm has a linear complexity, and they also demonstrate that the actual performance is reasonable in absolute terms, with a maximum off-line running time of about 50 seconds for the full workload of 200 million tag sets. As a rough comparison, consider that MongoDB requires about 33 seconds for a table of only 5 million sets, for which our partitioning algorithm runs in about 2 seconds.

Figure 3.14 shows the memory usage on the CPU (Host) and GPU sides. The Host memory is used almost exclusively for the key table, with only a small portion for the partition table and the buffers used for communication between the CPUs and the GPUs. The memory of the GPUs is used primarily for the tagset table, with a small fraction used for communication buffers.

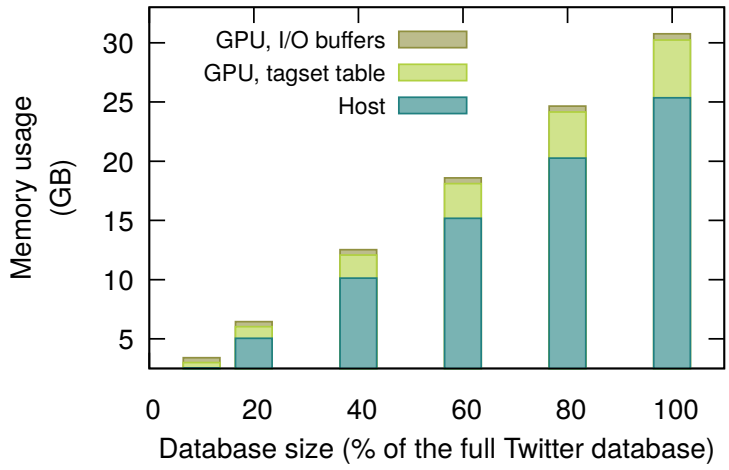

Figure 3.14. TagMatch Memory Usage (GB).

3.5.4 Comparison with MongoDB

This section evaluates the performance of subset query processing in MongoDB version 3.2.10. We treat MongoDB as a special case due to the significant difference in performance with TagMatch. In particular, since we could not use the full Twitter workload used in Section 3.5.3 due to the higher processing time and memory consumption of MongoDB, we construct and experiment with a scaled-down workload with a similar selectivity and with the same number of additional tags per query.

We configure MongoDB to store a database of sets of tags on a RAM disk in main memory. We also force MongoDB to index the entries of the database to improve the performance of the query process. We run MongoDB in two configurations, first as single server and then by sharding the database over multiple instances. In both settings, we use the Java API to connect and submit queries through a TCP socket (on localhost). We then use a single thread to submit asynchronous queries. We also experimented with multiple client connections. However, even though MongoDB can process queries from different connections in parallel, we did not observe any performance improvement.

Figure 3.15 shows the results of an experiments in which we compare TagMatch and MongoDB in the single-server setting with different database sizes, different numbers of tags per database set, and varying numbers of additional tags per query. Even with a small database of one million sets, MongoDB takes more than two seconds to process a single query, and the performance decreases significantly with the size of the database (notice the log scale), down to more

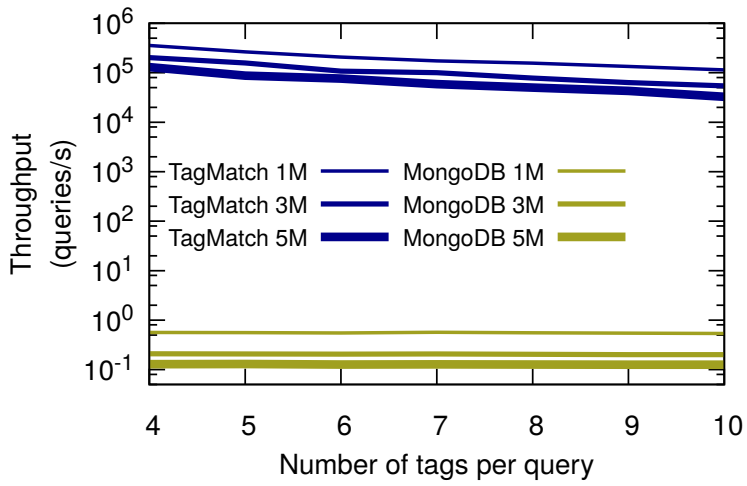

Figure 3.15. Comparison with MongoDB. Average Throughput for *match* with Different Number of Tags per Query.

than 10 seconds per query in the case of 5 million sets. Conversely, neither the number of tags in the database sets nor the number of additional tags in each query influence the overall performance of MongoDB, despite the fact that they both have a significant impact on the selectivity of the workload. In comparison, TagMatch can process more than 32,000 queries per second even in the most challenging scenario of 2-tags database-sets and 10-tags queries.

We also test a distributed deployment of MongoDB with a database sharded over multiple servers. In this setting, MongoDB sends each query to all the instances for processing on each individual database shard. We perform an experiment in this setting to evaluate the benefits and scalability of distribution and sharding. To minimize the network overhead, we run all MongoDB instances on the same physical machine, and only consider a relatively small deployment of up to 24 instances (the machine has CPU 24 cores and sufficient memory). We show the results for a database of 3 million entries, each containing 3 tags, and for queries of 6 tags. The results of this experiment, shown in Figure 3.16, demonstrate that sharding and distribution are clearly beneficial, and specifically that the throughput of MongoDB increases linearly up to 8 instances and overall by a factor of 3 with 24 instances. However, even assuming a perfectly linear scalability, MongoDB would require tens of thousands of server instances to reach the level of performance of TagMatch.

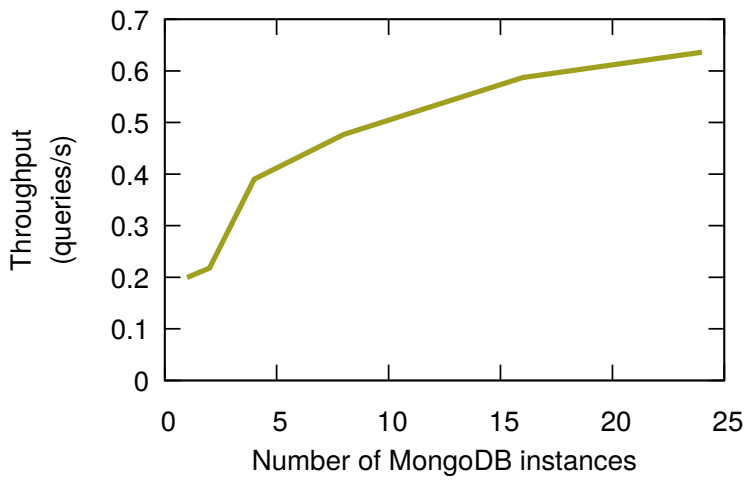

Figure 3.16. Scalability of MongoDB with Sharding. The Database Contains 3 Million Entries, with 3 Tags Each. Queries Contain 6 tags.

3.5.5 Experience with an Alternative Design

In the early stages of the development of TagMatch, we experimented with a design in which both the pre-process and subset-match phases would run entirely on GPUs. This architecture is technically feasible thanks to the "dynamic parallelism" of the newer NVidia GPUs.

With dynamic parallelism, it is possible to launch a new kernel from within a running kernel on the GPU. Thus, one can launch a kernel that performs the pre-process algorithm on a batch of input queries and that uses partition queues in the global memory of the GPU. When one of those queues fills up, the pre-process kernel can then invoke a new subset-match kernel on that queue directly from within the GPU.

This architecture is potentially advantageous, since it allows for numerous parallel subset-match kernels for different partitions. This approach also overcomes one of the factors that in part limits the performance of TagMatch, namely the need to transfer a single packet to the GPU potentially many times (one for each matching partition). Furthermore, the pre-process algorithm is also inherently parallelizable, and should therefore work well on a GPU.

And yet, the prototype we built was not that efficient. As it turns out, the GPU-only design works well when the vast majority of packets are filtered out in the pre-process phase, but not when many packets reach the subset-match phase. In this latter case, the pre-process algorithm must copy many queries into potentially many partition queues, which induces many atomic operations

and an almost random access pattern into the global (slow) memory of the GPU. Also, the GPU-only design still requires synchronization, as well as the transfer of some partial results between CPUs and GPUs, which further limits parallelism.

Chapter 4

An Ideal Routing Scheme for a Wireless Network Model

In our quest to develop an ideal routing scheme, with the properties shown in Table 1.1, we started this research with a simpler routing scheme where we put restrictions on both the communication and network models and focused our attention on the congestion property of the proposed routing scheme.

What we present in this chapter is developed chronologically earlier than other contribution of this thesis and we see it as an initial step that shaped our understanding of the challenges we face in designing a multicast routing scheme for ICN. This work is not the main focus of this thesis and is an expanded version of the paper we published in the proceedings of the thirteenth ACM international symposium on mobile ad hoc networking and computing (MobiHoc '12) with the title "*oblivious low-congestion multicast routing in wireless networks*" [21].

We propose a routing scheme to implement multicast communication in wireless networks. The scheme is oblivious, compact, and completely decentralized. It supports explicit addressing, so it is not itself a content-based scheme. Still, it is intended to support dynamic and diverse multicast requests typical of content-based communication in ICN. The scheme is built on top of a geographical routing layer. Each message is transmitted along the geometric minimum spanning tree that connects the source and all the destinations. Then, for each edge in this tree, the scheme routes a message through a random intermediate node, chosen independently of the set of multicast requests. The intermediate node is chosen in the vicinity of the corresponding edge such that congestion is reduced without stretching the routes by more than a constant factor. We first evaluate the scheme analytically, showing that it achieves a

theoretically optimal level of congestion. We then evaluate the scheme in simulation, showing that its performance is also good in practice.

4.1 Problem Setting

Content-based communication is inherently multicast, since implicit content-based addressing induces the transmission of a single message to multiple destinations (all interested receivers). In particular, while some multicast services are based on a few and relatively stable multicast groups (e.g., video streaming over IP multicast) and therefore work well with stable routing state, content-based communication is more demanding and more dynamic. This is because receiver interests may partially overlap, forming a large number of implicit groups, potentially a different one for each message.

We therefore consider a generic multicast primitive in which each message may induce a unique multicast request (m, s, T). This primitive allows a source node s to send a message m to a set of target nodes T. In particular, we consider this primitive within a wireless network. Our goal is to implement such a communication primitive through a routing scheme that is oblivious, compact, low-stretch, low-congestion, and also practical.

The scheme we propose is *oblivious* in the sense that how a request is routed does not depend on the set of requests and how the other requests are routed. We in fact prove that the scheme offers the best possible performance guarantees even in the presence of adversarial requests. The scheme is *compact* in the sense that it requires only limited state at each node, typically $O(\text{polylog}\, n)$ bits in a network of n nodes. The scheme is *low-stretch*, in the sense that the length of each path from a source to a target node, which roughly corresponds to the latency of each delivery, is optimal up to a small constant factor. The scheme is also *low-congestion*, in the sense that, for any given set of multicast requests, the maximum amount of traffic crossing a node is only a factor of $O(\log n)$ worse than with an ideal routing specifically optimized for that set of requests. Notice that this $O(\log n)$ factor for congestion is optimal for any oblivious scheme [14, 57], even for routing on 2-dimensional meshes. Lastly, the scheme is *practical* in the sense that the theoretical asymptotic behavior of the scheme can be realized in practice with good pre-asymptotic performance and small constants.

The scheme we propose is built on top of a geographical routing service whereby a message can be addressed to a given geographical location and therefore can be delivered, possibly through multiple hops, to the node that

is closest to that location. Such geographical schemes exist and are compact and achieve low-stretch both theoretically and in practice [51]. The choice of a geographical communication primitive implies that, in its most basic form, the routing scheme we propose is *name dependent*. This means that nodes must be identified by some kind of address dictated by the communication layer (in this case, the node's geographic coordinates). However, it is also possible to extend such a basic routing scheme to be name independent, by means of a lookup service that can also be implemented efficiently [2].

In summary, we start from a compact and low-stretch geographical routing substrate, which for a request (m, s, t) can deliver a unicast message m from a source s to a target destination t, and we use it to build a low-congestion oblivious multicast scheme that can serve requests of the type (m, s, T) and deliver m from a source s to a set of target destinations T. A simple way to implement such a multicast scheme would be to implement each multicast request (m, s, T) with a series of unicast requests (m, s, t_i) for each t_i in T. However, such a scheme incurs high congestion. Intuitively, this is the case when many destinations are close to each other, even without adversarial sets of requests and instead with sources and destinations distributed uniformly over multiple requests.

A standard way to achieve low congestion with an oblivious unicast scheme is to use randomization in what is known as Valiant's trick [80]. For a unicast request (m, s, t), first route m from s to a randomly chosen intermediate destination v, and then from v to t. However, in its basic form, this trick does not work well for arbitrary worst-case sets of requests and in particular it does not work well for multicast requests. Consider for example a request (m, s, T) in which the targets $t_i \in T$ are all clustered in a small region far away from the source s. Even with Valiant's trick, a series of (unicast) copies of m going from s to a target t_i in the cluster would induce high congestion in the small perimeter around the cluster, whereas an optimal routing strategy in that case would send one copy of m from s towards the cluster, and then it would duplicate m locally to all targets within the cluster.

The scheme we propose employs a local variant of Valiant's trick, and it does that within a routing strategy that avoids congestion in the case of multicast requests. At a high level, the scheme routes a multicast request (m, s, T) along the geometric minimum spanning tree that connects the source s and all the targets in T. Then, for each edge (u, v) on that tree, the scheme uses a variant of Valiant's trick by routing m from u to an intermediate point w_{uv} chosen randomly in the vicinity of the uv segment.

In the following we formally define this routing scheme, we then analyze its

theoretical properties, and evaluate it in practice using simulation. The theoretical analysis shows that, in terms of congestion, the scheme is competitive with an ideal (non-oblivious) scheme up to a factor of $O(\log n)$, which is known to be a lower bound for congestion in oblivious schemes. The simulation study shows that the scheme is also effective in practice, with limited congestion and stretch.

4.2 Related Work

Compared to classic wired networks, wireless ad hoc and sensor networks behave more dynamically. As a consequence, classical link-state routing protocols are often not well-suited for wireless networks and other, more reactive routing strategies are required. A standard way to do this is to combine flooding for route discovery with some caching techniques to reuse acquired routing information [18, 43, 68, 74]. While there is an abundant literature on wireless point-to-point routing, the work on wireless multicast is much less copious. In fact, Vershney claims that wireless multicast is still an important challenge [81]. Multicast protocols for wireless networks have been suggested, for example, by Royer and Perkins [75] or by Xie et al. [83].

Since the presence of wireless communication links is inherently related to the physical placement of nodes, if available, geometric information can be a powerful tool for routing. For geographic routing, it is typically assumed that all nodes are aware of their geographical position and the source node of a message knows the location of the destination. The simplest possible way to route a message that way is to proceed greedily by always forwarding a message to the neighbor closest to the destination [77]. While greedy routing is efficient in dense average-case scenarios, it might not always reach the destination. The first proposed geographic routing protocol that is guaranteed to reach the destination is face routing [48]. The delivery guarantees of the face routing protocol come at the cost of worse behavior in well-behaved settings. Therefore greedy and face routing have been combined to obtain average-case efficient protocols with guaranteed message delivery [16, 45, 51]. All these geographic routing protocols assume that the communication network is a unit disk graph. In this work, we extend this setting with non-uniform transmission ranges in a model similar to those proposed by others [13, 50].

To apply geographic routing, the source node of a message needs to know the location of the destination. A typical application is geocast, a variant of multicast, where all nodes in a certain geographical region have to be reached [58].

If location information of the destination is not available, geographic routing can be combined with a location service that allows to efficiently search for location information of other nodes [2, 29, 55].

All routing schemes described so far do not explicitly attempt to minimize the congestion that arises in the presence of a large number of routing requests. From an algorithmic point of view, congestion has mainly been considered in the context of oblivious routing, i.e., if each routing path is chosen independently. A seminal result by Valiant and Brebner [80] shows that in a hypercube, any permutation can be routed in $O(\log n)$ steps. The path selection is randomized and uses what is now known as Valiant's trick. Each message is first routed to a random intermediate node and from there to the destination. The technique has been applied in various other networks and in particular, it was shown by Kolman and Scheideler [46] that Valiant's trick can efficiently be used in a much more general setting. The existing work on oblivious routing culminated in a breakthrough paper by Räcke [70] that shows that there is an oblivious protocol that routes every set of routing requests with expected maximum node congestion within a logarithmic factor of the best corresponding multi-commodity flow solution. In light of a lower bound that even holds for 2-dimensional meshes, this is asymptotically optimal [14, 57]. Räcke's result also applies to multicast and could also be used for our wireless network model. However, the protocol state is rather heavy-weight to set up and maintain, and the given wireless setting is amenable to specialized and much more light-weight algorithms. Most closely related to our work are two papers by Busch et al. that describe algorithms for unicast in 2-dimensional meshes [20] and for geometric networks modeling dense wireless networks [19]. For unicast, this latter algorithm [19] achieves the same asymptotic bounds as the algorithm presented here. However, we believe that our randomized scheme based on Valiant's trick is somewhat simpler and easier to use. A recent survey on oblivious routing is also due to Räcke [71]. Other papers study congestion in the context of wireless network routing, but are less related to this work [31, 54, 69, 85].

4.3 Model and Definitions

We now formally state our assumptions about the communication network and its underlying geographic routing service.

Communication Network: We assume that n wireless network nodes are located in a bounded region in 2-dimensional Euclidean space. The nodes have

unique identifiers and we denote the set of nodes by V. For simplicity, we assume that the region is a square of side length L, however, the techniques work for any "reasonable" convex region. Further, we assume that nodes are aware of their position in the plane. This can be achieved by equipping nodes with GPS devices or through some localization service. Communication in the network is characterized by two positive parameters $r_C \leq r_I$ defining communication and interference radii. Whenever two nodes u and v are at Euclidean distance at most r_C, u and v can directly communicate with each other. If two nodes u and v are within distance r_I, they can cause interference to each other. Further, we assume that there is no direct communication or even interference between two nodes at distance more than r_I. We denote the ratio between r_I and r_C by $\rho := r_I/r_C$ and typically assume ρ to be a constant (independent of n). We assume that the $L \times L$-square containing the network is reasonably densely covered by nodes. Specifically, we assume that there is a parameter r_{cov} such that for every point in the $L \times L$-square, there is a network node within distance r_{cov}. We assume r_{cov} is relatively small, such that the requirement implies that the number of nodes is at least polynomial in L/r_I.

Geographic Routing: We assume that there is a geographic routing service in place, which nodes use for communicating with each other. More formally, a node u can send a message to an arbitrary (x, y) coordinate pair within the specified geometric region that contains the wireless network nodes (i.e., the side length L square). If a message is sent to (x, y), the routing service guarantees that the node closest to (x, y) (according to Euclidean distance) receives the message. We assume that nodes populate the complete given geometric region densely enough to enable routing on almost direct paths between all pairs of nodes. We use the following definitions:

Definition 4.3.1 (λ-Padded Path)**.**
A path $P = u_1, \ldots, u_k$ connecting coordinates (x, y) and (x', y') is λ-padded if all nodes u_i of P are within Euclidean distance at most $\lambda \cdot r_I$ from the line segment connecting (x, y) and (x', y') in the plane.

Definition 4.3.2 (σ-Sparse Path)**.**
A path $P = u_1, \ldots, u_k$ is called σ-sparse if no disk of diameter r_C contains more than σ nodes u_i of P.

We assume the geographic routing service induces λ_{pad}-padded, σ-sparse paths for some positive parameters λ_{pad} and σ. Note that this in particular implies that the node distribution is dense enough so that there is a node at distance at most $\lambda_{\text{pad}} r_I$ from every point (x, y) in the geometric region covered

by the network, i.e., $r_{\text{cov}} \leq \lambda_{\text{pad}} r_I$. Further note that the assumption that any two nodes within distance r_C are connected implies that nodes inside a disk of diameter r_C are fully connected and therefore, paths containing more than 2 nodes in such a disk can be shortened to contain at most 2 such nodes. Hence, if a λ_{pad}-padded path between (x, y) and (x', y') exists, then there is also a λ_{pad}-padded, 2-sparse path between the two points.

Typically, for relatively dense average-case networks, services based on greedy routing perform best. By construction, greedy routing always gives 2-sparse paths. Further, as shown in Section 4.7, it also gives good, $O(1)$-padded paths. For worst-case networks, geographic routing techniques [50, 51] can be used to find an $O(1)$-sparse, $O(\lambda)$-padded path, whenever a λ-padded path exists.

4.4 Problem Statement

Multicast Routing: We consider two variants of the multicast problem. A lower level geographic and a high-level name-based variant. In both cases, we are given r multicast requests $R_1, \ldots, R_r$ where request $R_i = (m_i, s_i, T_i)$ consists of a message m_i, a source node s_i and a set T_i of k_i destinations $t_{i,1}, \ldots, t_{i,k_i}$. We assume that s_i knows m_i and T_i and the objective is for s_i to send m_i to all destinations in T_i. In the case of the *geographic multicast problem*, each destination $t_{i.j}$ is given as a coordinate pair $(x_{i,j}, y_{i,j})$ and for all $i \in [r]$, message m_i has to be sent to the k_i actual network nodes closest to $(x_{i,1}, y_{i,1}), \ldots, (x_{i,k_i}, y_{i,k_i})$. In the more standard *name-based multicast problem*, each destination $t_{i,j}$ is given as a node identifier. As usual in the context, we assume that messages m_i are large compared to the size of T_i, so that the overhead of storing all destination information in the message header is negligible [5]. The geographic multicast problem is closely related to what is generally known as geocast [58]. Unlike specifying individual destinations, typically, the destinations are given by a geographic region to which a message has to be transmitted. We note that the geographic multicast service that we present can easily be adapted to efficiently work in such a scenario. In fact, in our communication model, sending to a geographic region can be modeled by sending to a dense enough set of destinations within the area.

Congestion: As discussed in Section 4.3, we assume that nodes at distance at most r_I can cause interference to each other. To model congestion, we assume that whenever a node u transmits, it causes interference at all nodes within

distance r_I from u. Let I_u be the set of nodes within Euclidean distance r_I from node u. Hence, whenever a node in I_u sends a message, it causes interference at node u and vice versa, whenever u transmits a message, it interferes with all nodes in I_u.

To satisfy a given multicast request $R_i = (m_i, s_i, T_i)$, message m_i has to be sent from s_i to all nodes in T_i along a subtree of the network. Given some algorithm $\mathcal{A}$, let $S_i^{\mathcal{A}}$ be the multiset of nodes that transmit message m_i in order to reach all destinations in T_i, i.e., $S_i^{\mathcal{A}}$ at least contains all the inner nodes of the tree along which m_i is sent to the destinations. Given a set of r multicast requests $R_1, \ldots, R_r$ and an algorithm $\mathcal{A}$, we define the congestion $\mathsf{cong}_u^{\mathcal{A}}$ of a node u and the maximum node congestion $\mathsf{cong}^{\mathcal{A}}$ of $\mathcal{A}$ as

$$\mathsf{cong}_u^{\mathcal{A}} := \sum_{i=1}^{r} |S_i^{\mathcal{A}} \cap I_u|, \quad \mathsf{cong}^{\mathcal{A}} := \max_{u \in V} \mathsf{cong}_u^{\mathcal{A}}. \tag{4.1}$$

Our main objective will be to minimize $\mathsf{cong}^{\mathcal{A}}$. Whenever it is clear from the context, we omit the superscript $\mathcal{A}$. In order to evaluate an algorithm, we intend to compare its behavior with the best possible maximum node congestion. Let $\mathsf{cong}^\star$ be the maximum node congestion of an optimal routing solution for the given requests $R_1, \ldots, R_r$. Consider a rectangle $\mathcal{R}$ with side lengths $w(\mathcal{R})$ and $h(\mathcal{R})$. We define $\mathsf{cut}(\mathcal{R})$ to be the set of requests R_i, $i \in [r]$ such that $\{s_i\} \cup T_i$ contains at least one node inside $\mathcal{R}$ and at least one node outside $\mathcal{R}$. To bound the optimal congestion $\mathsf{cong}^\star$, we introduce the following notion:

$$\mathsf{load}(\mathcal{R}) := \min \left\{ |\mathsf{cut}(\mathcal{R})|, \frac{\big|\mathsf{cut}(\mathcal{R})\big| \cdot r_I}{w(\mathcal{R}) + h(\mathcal{R})} \right\}. \tag{4.2}$$

The following lemma shows that asymptotically, $\mathsf{load}(\mathcal{R})$ is a lower bound on the best possible maximum congestion $\mathsf{cong}^\star$.

Lemma 4.4.1. *For every set of multicast requests $R_1, \ldots, R_r$ and every rectangle $\mathcal{R}$, we have* $\mathsf{cong}^\star = \Omega\big(\mathsf{load}(\mathcal{R})\big)$.

Proof. Consider a multicast request R_i for which $\{s_i\} \cup T_i$ contains at least one node inside $\mathcal{R}$ and at least one node outside $\mathcal{R}$. Further, let B be the geometric area defined by all points within distance r_I of the boundary of $\mathcal{R}$. In order to satisfy request R_i, a message has to be sent into or out of $\mathcal{R}$ and therefore at least one node in B has to transmit a message.

Consider a maximal independent set S of the graph defined by the nodes V_B that lie inside B and edges $\{u, v\}$ whenever u an v are at Euclidean distance

at most r_I. Whenever a node in B transmits a message, it causes congestion at some node in S. Further, since nodes in S are within distance more than r_I, the number of nodes in S is at most $O\big(1 + (w(\mathcal{R}) + h(\mathcal{R}))/r_I\big)$. Hence, by the pigeonhole principle, for every solution for the given multicast problem, some node in S has congestion at least $\Omega(\mathsf{load}(\mathcal{R}))$. □

4.5 Geometric Multicast

Our algorithm consists of two components, which together allow to multicast a message to a set of geographical destinations based on an underlying geographic routing service as discussed in Section 4.3. At the core is an oblivious geographic point-to-point routing protocol with asymptotically optimal congestion properties. A multicast request is then routed on a tree by applying the point-to-point scheme.

We first describe the routing scheme to send a message from a node u to a geographical destination (x, y). The point-to-point routing algorithm is based on Valiant's classical trick of reducing overall congestion by routing messages through a randomly chosen intermediate node. To deal with worst-case collections of routing requests and to guarantee a bounded stretch factor for the routing paths, we choose the random intermediate point dependent on the source and target positions of the routing request. Specifically, a message from a node u at position (x_u, y_u) to location (x, y) is routed as follows.

1. If the Euclidean distance of (x, y) from (x_u, y_u) is at most $r_C/2$, the node closest to (x, y) is either u itself or a neighbor v of u. In that case, u directly sends the message to v.

2. Otherwise, node u chooses a random intermediate position (x_r, y_r) as follows. First, u chooses two uniform random angles $\alpha, \beta \in [0, \pi/3]$. The point (x_r, y_r) is then chosen such that the line segments from u to (x_r, y_r) and from u to the destination position (x, y) enclose an angle of α and the line segments from (x, y) to (x_r, y_r) and from (x, y) to u enclose an angle of β. There are two points (x_r, y_r) for which this is true (one to the left and one to the right of the line connecting source and destination). Node u randomly chooses one of the two points as (x_r, y_r).

   Using the underlying geographic routing protocol, the message is then routed from u to the node w closest to (x_r, y_r) and afterward from node w to the destination position (x, y).

The choice of the random point (x_r, y_r) is also illustrated in Figure 4.1. Note that (x_r, y_r) is chosen such that the geometric distances from (x_r, y_r) to u and (x, y) are at most as large as the distance between u and (x, y).

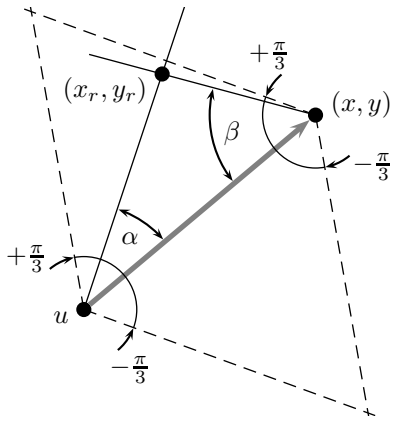

Figure 4.1. Choice of Intermediate Node

Based on the described scheme for point-to-point communication, we can now build the multicast routing protocol on top of it. For a given geographic multicast request $R_i = (m_i, s_i, T_i)$, let (x_i, y_i) be the position of the source node s_i and let $P_i = \{(x_i, y_i)\} \cup T_i$ be the set of points of the multicast request R_i. We first construct a geometric tree spanning all the points in P_i and then use the point-to-point routing algorithm to send m_i along all the edges of the constructed spanning tree. There are different ways to choose the geometric spanning tree of the points in P_i. In terms of total routing cost, the best choice would be to choose a minimum Steiner tree w.r.t. Euclidean distances. Note that the Euclidean Steiner tree problem is NP-hard. However, there is a polynomial-time approximation scheme and thus the problem can be approximated arbitrarily well [10]. Still, since we would like our algorithm to be as simple as possible, and also since asymptotically it does not make a difference, we use the Euclidean minimum spanning tree (MST) to connect the points in P_i. Such a tree can be computed locally by the sender with an efficient algorithm.

The message m_i is sent along the edges of the Euclidean MST of P_i in a straightforward manner. The tree is directed from the source s_i at (x_i, y_i) towards the destinations T_i and slightly adapted in the following way. As long as there is a directed path u, v, w such that the Euclidean distance between u and w is at most $r_C/2$, node w is attached directly to u instead of v. This allows to reach close-by nodes by local broadcast where possible.

For each (directed) edge $\big((x,y),(x',y')\big)$, a message is sent from the node closest to (x,y) to the node closest to (x',y') by using the point-to-point routing scheme described above. Assume that a node w representing a node (x,y) in the tree needs to send messages to different neighbors (x',y') in the MST. If some of the neighbors (x',y') are at distance at most $r_C/2$ from the position of w, w sends one broadcast message to all neighbors to reach the nodes closest to these tree neighbors. For all other tree neighbors, the message is sent by using the randomized point-to-point routing scheme, i.e., the message for each edge is routed via a random intermediate point as described above.

4.5.1 Analysis

Recall that for the analysis, we assume that for every point in the $L \times L$-square containing the network, there is an actual node within distance r_{cov}. Further when routing from a node at point (x,y) to a node at point (x',y'), the underlying geographic routing service generates λ_{pad}-padded, σ-sparse paths. For the analysis, we require a technical lemma bounding the number of local long edges of an MST in the Euclidean plane.

Lemma 4.5.1. *Let T be an Euclidean MST of a set of points $X \subseteq \mathbb{R}^2$ and consider a circle $C \subseteq \mathbb{R}^2$ of radius r. The number of edges $\{p,q\}$ of T of length at least $3r$ such that $|\{p,q\} \cap C| = 1$ (i.e., edge $\{p,q\}$ connects a point inside C with a point outside C) is at most 7.*

Proof. Consider two edges $\{p,q\}$ and $\{p',q'\}$ of length at least $3r$ such that $p,p' \in C$, $q,q' \notin C$. Let c be the center of the circle C and let θ be the angle that is enclosed by the rays cq and cq'. Let d_{cq}, $d_{cq'}$, and $d_{qq'}$ be the Euclidean distances between c and q, c and q', as well as q and q', respectively. By the law of cosines, we have

$$\cos\theta = \frac{d_{cq}^2 + d_{cq'}^2 - d_{qq'}^2}{2 \cdot d_{cq} \cdot d_{cq'}}. \tag{4.3}$$

Our goal is to upper bound the above expression and therefore to get a lower bound on the angle θ. Because the edges $\{p,q\}$ and $\{p',q'\}$ have length at least $3r$ and because p and p' lie in the circle C, it follows that

$$d_{cq} \geq 2r \quad \text{and} \quad d_{cq'} \geq 2r. \tag{4.4}$$

Since both p and p' are inside C, their distance is at most $2r$. Because $\{p,q\}$ and $\{p',q'\}$ are edges of the MST T and because we assume that their length

is at least $3r$, $d_{qq'}$ has to be at least as large as the length of the longer of the two edges $\{p, q\}$ and $\{p', q'\}$. W.l.o.g., assume that $d_{cq} \geq d_{cq'}$. We then get

$$d_{qq'} \geq \max\{3r, d_{cq} - r\}. \tag{4.5}$$

We obtain an upper bound on $\cos\theta$ by maximizing the right-hand side of (4.3) subject to $d_{cq} \geq d_{cq'}$ and Inequalities (4.4) and (4.5). For fixed values of d_{cq} and $d_{qq'}$, the r.h.s. of (4.3) is a concave function of $d_{cq'}$ and is thus maximized either for $d_{cq'} = 2r$ or for $d_{cq'} = d_{cq}$.

- $\boldsymbol{d_{cq'} = 2r}$: In that case, the r.h.s. of (4.3) is monotonically increasing in d_{cq} and therefore maximized for $d_{cq} = d_{qq'} + r$. We then get

  $$\cos\theta \leq \frac{(2r)^2 + 2rd_{qq'} + r^2}{4 \cdot (rd_{qq'} + r^2)} \leq \frac{11}{16}.$$

  The second inequality follows from $d_{qq'} \geq 3r$.

- $\boldsymbol{d_{cq'} = d_{cq}}$**:** In the second case, we get

  $$\cos\theta = 1 - \frac{d_{qq'}^2}{2d_{cq}^2}.$$

  The above expression gets large if $d_{qq'}$ is as small as possible and d_{cq} is as large as possible. It is maximized for $d_{qq'} = d_{cq} - r = 3r$, in which case we obtain

  $$\cos\theta = 1 - \frac{(3r)^2}{2(4r)^2} = \frac{23}{32}.$$

Combining the two cases, we therefore get $\cos\theta \leq 11/16$ which implies that $\theta > 0.812 > 2\pi/8$. □

In the following, let T_i, $1 \leq i \leq r$ be the Euclidean MST corresponding to multicast request R_i and let E_i be the directed edges of T_i, where each edge is directed away from the source s_i of R_i (i.e., in the direction in which a message has to be sent). Let $E_{\text{MST}} = \bigcup_{i=1}^{r} E_i$ be the set of all directed MST edges. For a region $A \subseteq \mathbb{R}^2$ in the plane, let $E_{\text{MST}}^{\nearrow}(A)$ be the set of directed edges $(p, q) \in E_{\text{MST}}$ for which $p \in A$ and let $E_{\text{MST}}^{\swarrow}(A)$ be the set of directed edges $(p, q) \in E_{\text{MST}}$ for which $q \in A$. For each long enough edge $(p, q) \in E_{\text{MST}}$, two messages are sent by using the underlying geographic routing service, one message from p to a random intermediate destination and one message from the intermediate destination to q. Let $\mathcal{M}_{\text{out}}(A)$ be the set of messages sent from p to the random intermediate destination for an edge $(p, q) \in E_{\text{MST}}^{\nearrow}(A)$. Further, let $\mathcal{M}_{\text{in}}(A)$ be the set of messages sent from the random intermediate node to q for an edge $(p, q) \in E_{\text{MST}}^{\swarrow}(A)$.

Lemma 4.5.2. *Consider a square S with side length s and let v be a node at distance at least $d \geq 3s + 2r_{\text{cov}} + (\lambda_{\text{pad}} + 1)r_I$ from S. The expected congestion at node v caused by messages in $\mathcal{M}_{\text{in}}(S)$ and $\mathcal{M}_{\text{out}}(S)$ is at most $O\left((\lambda_{\text{pad}} + 1) \cdot \sigma \cdot \rho^2 \cdot \mathsf{load}(S)\right)$.*

Proof. Let us first consider a message $m \in \mathcal{M}_{\text{out}}(S)$ corresponding to some edge $(p, q) \in E^{\nearrow}_{\text{MST}}(S)$. The message m is sent from the node u closest to p to a random intermediate point (x_r, y_r). Assume that the coordinates of u are (x_u, y_u). Because $p \in S$, (x_u, y_u) is at distance at most r_{cov} from S. Further, by the way the random point (x_r, y_r) is chosen, the distance from u to (x_r, y_r) is upper bounded by the distance from u to q.

Message m only causes congestion at node v if the underlying geographic routing service sends the message through a node within distance r_I from v. Because we assume that the geographic routing paths are λ_{pad}-padded, this can be the case if v is within distance $r_I(1+\lambda_{\text{pad}})$ from the line segment connecting (x_u, y_u) and (x_r, y_r). Consequently, because the distance from v to S is at least $3s + 2r_{\text{cov}} + (\lambda_{\text{pad}} + 1)r_I$, the distance between u and (x_r, y_r) and therefore also the distance between u and q needs to be at least $3s + r_{\text{cov}}$. Because u is at distance at most r_{cov} from S, this implies that the edge (p, q) has length at least $3s$.

Further, recall that the line from u to (x_r, y_r) is at a random angle $\alpha \in [-\pi/3, \pi/3]$ from the line uq. Message m causes interference at node v only when α is such that the line connecting u and (x_r, y_r) passes within distance $(\lambda_{\text{pad}} + 1)r_I$ from v. Let β be the angle between line uv and the line connecting u with (x_r, y_r). The angle β is also a uniform random angle from some interval $[\beta_0, \beta_1]$ of length $2\pi/3$. Message m can cause interference at v if $|\beta| \leq \pi/2$ and $\ell \cdot \sin\beta \leq (\lambda_{\text{pad}} + 1)r_I$, where $\ell \geq 3s$ is the distance between u and v. Using $|\sin\beta| \leq |\beta|$, we get

$$|\beta| \leq \frac{(\lambda_{\text{pad}} + 1)r_I}{\ell} \leq \frac{(\lambda_{\text{pad}} + 1)r_I}{3s}.$$

Let $C_{m,v}$ be the event that message m causes congestion at v. The probability for this to happen is at most

$$\mathbb{P}(C_{m,v}) \leq \frac{2(\lambda_{\text{pad}} + 1)r_I)}{3s} \cdot \frac{1}{2\pi/3} = \frac{(\lambda_{\text{pad}} + 1)r_I}{\pi \cdot s}. \tag{4.6}$$

We define $X_{m,v}$ to be the random variable that counts the amount of congestion caused by m at v. Hence, $X_{m,v}$ is the number of nodes in the r_I-neighborhood of v that transmit a message while sending m from u to (x_r, y_r). Clearly $X_{m,v}$

can only be positive if the event $C_{m,v}$ occurs. In this case, the value of $X_{m,v}$ is at most $O(\sigma\rho^2)$ because we assume that the paths created by the geographic routing service are σ-sparse and a disk of radius r_I can be covered with $O(\rho^2)$ disks of diameter r_C. Let $X = \sum_{m \in \mathcal{M}_{\text{out}}(S)} X_{m,v}$ be the congestion at v caused by messages in $\mathcal{M}_{\text{out}}(S)$. By linearity of expectation, we have

$$\mathbb{E}[X] = O\left(\frac{(\lambda_{\text{pad}} + 1)r_I}{s} \cdot \sigma\rho^2 \cdot |\mathcal{M}_{\text{out}}(S)|\right).$$

To bound $\mathbb{E}[X]$, it therefore remains to bound the number of messages in $\mathcal{M}_{\text{out}}(S)$. We have seen that each message $m \in \mathcal{M}_{\text{out}}(S)$ corresponds to some MST edge (p, q) of length at least $3s$. Consider the circle C of radius $s/\sqrt{2}$ that encloses the square S. Since $p \in S$ and q is at distance at least $3s$, we have $p \in C$ and $q \notin C$. Hence, by Lemma 4.5.1, for each MST, there are at most 7 such edges of length at least $3s/\sqrt{2} < 3s$. Only multicast requests that contribute to $\mathsf{load}(S)$ can have MST edges with one node inside S and one node outside S. Further, for every such multicast request there are at most 7 edges in $\mathcal{M}_{\text{out}}(S)$. The expected congestion at v created by nodes in $\mathcal{M}_{\text{out}}(S)$ can therefore be upper bounded as

$$\mathbb{E}[X] = O\left((\lambda_{\text{pad}} + 1) \cdot \sigma \cdot \rho^2 \cdot \mathsf{load}(S)\right). \tag{4.7}$$

The situation for the messages in $\mathcal{M}_{\text{in}}(S)$ is almost symmetric. The messages are sent from the random intermediate destination (x_r, y_r) to a position inside S. However, the actual node sending the message might be at distance r_{cov} from (x_r, y_r), therefore we must accordingly adjust the angles for which there is congestion at node v. Instead of the value obtained in (4.6), the probability of $C_{m,v}$ can now be upper bounded by $\mathbb{P}(C_{m,v}) \leq \frac{r_{\text{cov}} + (\lambda_{\text{pad}}+1)r_I}{\pi \cdot s}$. Because $r_{\text{cov}} \leq \lambda_{\text{pad}} r_I$, this does not change anything asymptotically, and the congestion from messages in $\mathcal{M}_{\text{in}}(S)$ can also be upper bounded by the value given in (4.7). The claim of the lemma therefore follows Lemma 4.4.1. □

We are now ready to prove the main theorem of this section, showing that the expected maximal congestion induced by our geographic multicast algorithm is within a logarithmic factor of the optimal and therefore asymptotically best achievable for any oblivious algorithm [14, 57].

Theorem 4.5.3. *When using the described geographic multicast algorithm to route a given set of geometric multicast requests, the expected congestion at any node v is at most*

$$O\left(\left((\lambda_{\text{pad}} + 1) \cdot \log n + \lambda_{\text{pad}}^2\right) \cdot \sigma\rho^2 \cdot \mathsf{cong}^\star\right).$$

Proof. The multicast algorithm described at the beginning of Section 4.5 sends two kinds of messages. Most messages are messages sent through the underlying geographic routing layer. In addition, messages to local neighbors are sent by direct local broadcast. Node v can be affected by local broadcast messages only if they are sent by nodes within distance r_I from v. By adapting the MST structure and contracting paths of total length at most $r_C/2$, it is guaranteed that for each multicast request the number of local broadcast messages in each r_C-neighborhood is $O(1)$. Such messages must be sent by a node within range r_C. Hence, the total congestion at nodes within distance $r_I + r_C$ of v has to be within a constant factor of the congestion caused by local broadcast messages at v. Hence, for every multicast solution, there must be some node w close to v with congestion at least a constant times the congestion caused by local broadcast messages at v.

Let us therefore consider the congestion caused by messages that are sent through the underlying geographic routing layer. Note that all these messages correspond to an MST edge of length at least $r_C/2$ and they all either go from an MST node to a random intermediate destination or from a random intermediate destination to an MST node. We partition the $L \times L$-square containing the network into two parts, an area containing nodes close to v and an area with nodes far away from v. Specifically, we consider a square Q of side length $6r_{\text{cov}} + 3(\lambda_{\text{pad}} + 1)r_I = O\big((\lambda_{\text{pad}} + 1)r_I\big)$ and the area $\overline{Q}$ outside Q.

The area $\overline{Q}$ can be covered with $O\big(\log L/((\lambda_{\text{pad}} + 1)r_I)\big) = O(\log L/r_I)$ squares S_i of side length s_i such that the distance of square S_i to v is at least $3s+2r_{\text{cov}}+(\lambda_{\text{pad}}+1)r_I$ as follows. The area right around Q is covered with $O(1)$ squares of side length at most $\big(2r_{\text{cov}}+(\lambda_{\text{pad}}+1)r_I\big)/3$ such that Q together with these squares cover a larger square around v. The additional squares can be iteratively placed in the same way around the growing center square such that side length of the squares grows exponentially with the number of layers. By Lemma 4.5.2, for each of the squares S_i covering $\overline{Q}$, the expected congestion from messages in $\mathcal{M}_{\text{out}}(S_i)$ and $\mathcal{M}_{\text{in}}(S_i)$ is at most $O\big((\lambda_{\text{pad}} + 1)\sigma\rho^2\mathsf{cong}^\star\big)$. Hence, the expected congestion from messages sent from a node in $\overline{Q}$ to a random intermediate destination and from messages sent from a random intermediate destination to a node in $\overline{Q}$ is at most

$$O\left((\lambda_{\text{pad}} + 1) \cdot \sigma\rho^2 \cdot \log n\right) \cdot \mathsf{cong}^\star. \tag{4.8}$$

Recall that we assume r_{cov} is small enough and thus the node density is large enough such that n is at least polynomial in L/r_I and thus $\log(L/r_I) = O(\log n)$.

To prove the lemma, it remains to bound the congestion from messages sent from a node in Q to a random intermediate destination or from a random intermediate destination to a node in Q. Let M be the set of such messages. Because we assume that the geographic routing service produces σ-sparse paths and because the r_I-neighborhood of v can be covered by $O(\rho^2)$ disks of diameter r_C, the congestion from each message in M is at most $O(\sigma\rho^2)$. Hence, the congestion at v from messages in M is at most $O(|M|\sigma\rho^2)$.

Every message in M corresponds to an MST edge of length more than $r_C/2$ and there are at most 2 messages in M for each such MST edge. Further, for a particular multicast request, the number of MST edges of length more than $r_C/2$ with one node in Q is linear in the number of nodes in Q and at pairwise distance more than $r_C/2$. Hence, to serve all destinations in Q, in an optimal multicast protocol, nodes in Q or within distance r_C of Q need to transmit at least $\Omega(|M|)$ times. The square Q and its r_C-neighborhood can be covered with $O\left((\lambda_{\text{pad}}+1)^2\right)$ disks of diameter r_I. Each message that is transmitted by a node inside this area causes congestion at all nodes in at least one of these diameter r_I disks. Hence, by the pigeonhole principle, some node in Q or its r_I-neighborhood has congestion at least $\Omega\big(|M|/(\lambda_{\text{pad}}+1)^2\big)$. Thus, the congestion at v caused by messages in M can be upper bounded by

$$O\left(\left(\lambda_{\text{pad}}^2+1\right)\cdot\sigma\cdot\rho^2\cdot\mathsf{cong}^\star\right). \tag{4.9}$$

Since the congestion caused by local broadcast messages is within a constant factor of the optimal congestion, (4.8) and (4.9) together imply the claim of the theorem. □

Remarks: If the ratio $\rho = r_I/r_C$ and the parameters λ_{pad} and σ specifying the quality of the underlying geographic routing service are constants independent of n, the statement of the theorem simplifies. The theorem shows that in this case, the maximal expected node congestion of our multicast algorithm is within a factor $O(\log n)$ of the optimal maximum node congestion. Note that it is well known that this is the best achievable bound for oblivious routing. Further, since congestion contributions from different multicast requests are independent, a standard Chernoff argument shows that the bound of Theorem 4.5.3 does not only hold in expectation, but also with high probability. Finally, we would like to point out that within the quality guaranteed by the underlying routing layer, our multicast protocol produces routing paths and trees that are within a constant factor of the optimal.

4.6 Name-Based Multicast

The multicast protocol discussed in Section 4.5 allows to efficiently (in terms of congestion and stretch) multicast messages if the source node of a multicast request knows the positions of all the destinations. In many cases, information about the positions of destinations is not available to the node disseminating some information. In this case, a geographic routing service can be used in conjunction with a location service that allows to query the positions of nodes [2, 29, 55]. In the following, we sketch how to apply the LLS location service ([2]) to our context, and we show that, if for each multicast request the destination positions can be obtained with a small number of queries to the location service, then the expected maximal congestion of looking up the destination coordinates is within a constant factor of the expected maximal congestion incurred by multicasting the messages.

Let us first briefly discuss how LLS works. We describe the most basic variant of the scheme. (The authors also present a more involved scheme that takes into account update costs when nodes are moving [2].) LLS is essentially a geometric, distributed hash table. Assume that we want to store the location information for node v with identifier id_v. We assume that there is a hash function h that assigns a coordinate $h(\mathrm{id}_v) = \big(h_x(\mathrm{id}_v), h_y(\mathrm{id}_v)\big)$ in the $L \times L$-square to each node v. Using the position $h(\mathrm{id}_v)$, we define a hierarchical tiling of the plane into squares of exponentially decreasing sizes. The corners of the squares of level $\ell = 0, 1, 2, \ldots$ of the tiling are at positions $\big(h_x(\mathrm{id}_v) + i \cdot L/2^\ell, h_y(\mathrm{id}_v) + j \cdot L/2^\ell\big)$ for integers $i, j \in \mathbb{Z}$. On every level ℓ, the position information of v is stored at the four corners of the tile that contains v. Starting from the position of v in order of decreasing levels, v's information is stored in a spiral-like fashion.

To look up the coordinate information for some node v with identifier id_v, the protocol searches in the same spiral-like fashion. Assume that node u searches for v's position information. For each level ℓ, node u queries the four corners of the tile containing u in the tiling defined by $h(\mathrm{id}_v)$. The search is done in the order of decreasing ℓ, i.e., by going from small tiles to large tiles, which forms a spiral that is shown to hit a node that stores the information about v with asymptotically optimal cost [2]. The following is a list of the most important properties of the scheme for our purposes:

1. If a node u looks up the information of some node v, the distance that has to be traversed for the search is proportional to the Euclidean distance of u and v.

2. A search for node v starting at node u follows an exponentially growing spiral. The exact paths visited during the search are determined by the position $h(\mathrm{id}_v)$. Assuming that the hash function $h(\mathrm{id}_v)$ leads to a uniformly distributed position for the origin of the coordinate system defining the tiling, it can be shown that a search from node u causes interference at a node at distance d with probability proportional to $(\lambda_{\mathrm{pad}}+1)r_I/d$. Here, we assume that the search messages are sent through the geographic routing layer described in Section 4.3.

3. Assuming that the distribution of nodes is sufficiently dense, the scheme is compact. Each node only needs to store the position information of a logarithmic number of other nodes.

The next theorem shows that if at most κ look-ups are necessary for each multicast request, the expected look-up cost is asymptotically upper bounded by the expected cost for multicasting all message using our algorithm using the geometric protocol of Section 4.5. For the theorem, we assume that the hash function h leads to uniformly distributed positions $h(\mathrm{id}_v)$ that are independent of the given multicast requests. Due to lack of space, we only give a very rough sketch of the proof of the theorem.

Theorem 4.6.1. *If each multicast request requires to look up at most κ positions, at every node v, the expected congestion caused by all look-ups is at most*

$$O\left(\kappa \cdot \left((\lambda_{\mathrm{pad}}+1)\cdot \log n + \lambda_{\mathrm{pad}}^2\right)\cdot \sigma\rho^2 \cdot \mathsf{cong}^\star\right).$$

Sketch. The proof follows a similar reasoning to the one in Lemma 4.5.2 and Theorem 4.5.3, where the congestion of the geometric multicast algorithm is analyzed. According to the first property of LLS listed above, a search from a node u for a node v stays within distance $O(d(u,v))$ of u, where $d(u,v)$ is the Euclidean distance between u and v. Let us therefore assume that all the κ searches of the source s_i of some multicast request R_i stay within distance $c \cdot d(s_i, t_i)$, where t_i is the destination of request R_i that is farthest away from s_i.

Let us first consider the congestion at v caused by multicast requests with a source node that is relatively far away from v. Consider a square Q of side length d that is at distance at least $2c \cdot d + (\lambda_{\mathrm{pad}}+1)r_I$ from v. Assume that the source node s_i of multicast request R_i is inside Q. For a search of s_i to contribute to the congestion at node v, the farthest destination of R_i needs to be at least at distance $2d$ from s_i. Hence, R_i is a multicast request

that has the source node in Q and at least one destination node outside Q and R_i therefore contributes to $\mathsf{load}(Q)$ of Q. By the second property of LLS described above, the probability that a search of s_i causes congestion at v is at most $O((\lambda_{\text{pad}} + 1)r_I/d)$ and therefore by a similar argument as in the proof of Lemma 4.5.2, the expected total congestion at v from searches of source nodes in Q can be upper bounded by

$$O\left(\kappa \cdot (\lambda_{\text{pad}} + 1) \cdot \sigma \cdot \rho^2 \cdot \mathsf{load}(Q)\right).$$

By Lemma 4.4.1, this is within a factor $O\big(\kappa(\lambda_{\text{pad}} + 1)\sigma\rho^2\big)$ of the optimal maximal node congestion. As in the proof of Theorem 4.5.3, the congestion caused by source nodes at distance at least $3(\lambda_{\text{pad}}+1)r_I$ from v can be bounded by $O(\log n)$ times the above value because that part of the network can be covered with $O(\log n)$ squares to which the above argument can be applied. Also for the congestion from searches of sources within distance $3(\lambda_{\text{pad}} + 1)r_I$ from v, a similar argument to the one in the proof of Theorem 4.5.3 can be applied. Together, the bounds imply the statement of the theorem. □

4.7 Simulation Analysis

We now evaluate our routing scheme through simulation. This experimental analysis is intended to assess the performance of the scheme in practice, and also to characterize the effects of specific variants and parameters of the scheme itself as well as of the underlying geographical routing service. We consider three high-level research questions: (1) How does the scheme perform with various underlying routing algorithms? (2) How does the scheme perform with various selections of the random intermediate point? (3) How does the scheme perform in general under various workloads?

We first describe the implementation of the scheme and the underlying routing, and then present the simulation analysis.

4.7.1 Variants of the Routing Algorithms

We implemented two variants of the selection of the random intermediate point. The first variant corresponds exactly to the algorithm we describe and analyze formally in Section 4.5 and that is illustrated in Figure 4.1. This variant is parameterized by the range from which the source chooses the two random angles α and β that determine the intermediate point (x_r, y_r). In particular, we analyze the scheme when α and β are chosen uniformly in the ranges $[0, \pi/3]$,

$[0, \pi/4]$, and $[0, \pi/6]$. Intuitively, wider angles would disperse traffic and therefore reduce congestion, at the expense of slightly longer paths and therefore worse total traffic.

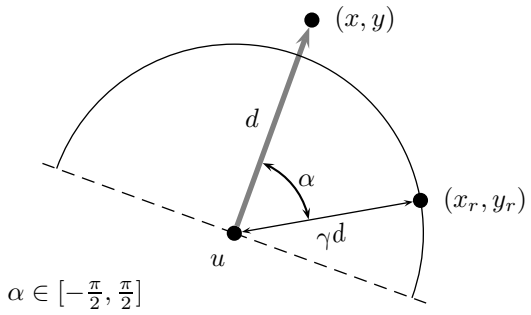

Figure 4.2. Alternative Selection of Intermediate Node

The second variant, illustrated in Figure 4.2, is a bit different: u selects an intermediate point (x_r, y_r) uniformly on a circular arc with center in u and radius $\gamma \cdot d(u, (x, y))$, where γ is a parameter of this method, and is between 0 and 2.

We also test our scheme with various underlying routing algorithms. Recall that the geographical routing layer sends a message from a source node u to the node closest to the destination (x, y). The algorithms we consider are:

Grd Greedy routing. Each node v forwards the message to the next-hop neighbor w that is the closest to the destination (x, y).

GSP Geometric shortest path. The path between u and (x, y) is minimal in terms of geometric length.

DSP Hop-count (or "Dijkstra") shortest path. The path between u and (x, y) is minimal in terms of number of hops.

GrdRnd1 A randomized variant of greedy routing. In this case a node v forwards a message to a next-hop neighbor w chosen uniformly among the ones that advance towards the destination by at least half of the communication radius r_C.

GrdRnd2 Another randomized variant of greedy routing. A node v forwards a message to a next-hop neighbor w chosen uniformly among the ones that are within half of the communication radius r_C from neighbor $\overline{w}$, which is the closest to the destination.

4.7.2 Experimental Setup and Parameters

We simulate a network of 80000 nodes spread uniformly over a square area of 100×100 units of length. (We also experimented with lower densities, obtaining consistent results that we do not report here for lack of space.) We set the communication radius to be equal to the interference radius ($r_C = r_I$) and we run simulations with $r_C = 1$ and $r_C = 2$ units of length. These settings correspond to a network that is dense enough to guarantee connectivity and to satisfy the more specific requirements of the underlying geographic routing, namely that it guarantees λ_{pad}-padded paths for a small constant λ_{pad}.

r_C	**GSP**	**DSP**	**Grd**	**GrdRnd1**	**GrdRnd2**
1	3.677	10.395	6.169	12.589	8.213
2	0.537	3.859	2.913	4.889	3.920

Table 4.1. $\boldsymbol{\lambda_{\text{pad}}}$ in Practice

Table 4.1 shows the actual values for λ_{pad} for all five geographical routing algorithms. These values were computed over 10000 randomly selected paths. Notice that these are *maximum* values (as per the definition of λ_{pad}-padded path) but at the chosen density the *average* distance between a routing path and a straight line between source and destination is much smaller. For the sake of brevity, in the rest of the of this chapter we discuss only the simulation with $r_C = 1$.

Workloads

We consider two classes of scenarios for multicast requests. One, which we denote as *uniform* in which requests involve sources and destinations chosen uniformly over the whole network, and one, which we denote as *in-line*, in which sources and destinations are chosen on a line, or more specifically on a narrow band in the middle of the network. The first class is intended to represent a generic traffic load. The second class is intended to represent a worst-case scenario for congestion. We also experimented with absolute worst-case workloads in which all requests are between the same source and the same destination. We initially show some results for all three cases for illustrative purposes, but then we focus on the *uniform* and *in-line* only because the third class is not very informative, since it incurs unavoidable congestion around the source and destination nodes.

Analysis

Figure 4.3 shows three "heat-map" graphs representing one simulation run for each of the three classes of workloads, respectively. The graphs represent the square region covering the simulated network. Each point in the graph represents a node in the network whose color represents the total traffic (number of wireless transmissions) affecting that node, which corresponds to *load* or *congestion* of that node.

In our analysis, we refer to a fixed set of all independent simulation parameters as a *scenario*. Thus, in a scenario we simulate all nodes running the same configuration of the geographic routing and the same configuration of our multicast routing scheme. We then simulate 1000 multicast requests, each with a fixed number of destinations chosen according to one of the scenario classes (*uniform* or *in-line*).

For each scenario we run 50 simulations to account for the variability that is due to the randomized nature of our scheme and possibly of the underlying routing. Then, for each node we compute the average load over the 50 runs, obtaining an approximation of the expected load of that node for that particular scenario. We then compute the *network congestion* as the maximum over all nodes of the per-node expected load. This is the primary metric of interest in this simulation analysis.

In summary, to answer our evaluation questions, we explore scenarios covering all combinations of the following parameters:

Intermediate point selection: type of algorithm and parameters used to select the intermediate point. We use the angle-based selection with bounds $\pi/3$, $\pi/4$, and $\pi/6$ denoted with *T60*, *T45*, and *T30*, respectively. We then use the circular-arc selection with distance multiplier $\gamma = 0, 0.5, 1, 1.5, 2$, which we denote as *C0*, *C0.5*, *C1*, *C1.5*, and *C2*. Notice that *C0* corresponds to using a deterministic straight-line routing scheme. This degenerate case is useful for comparison.

Geographical routing: type of algorithm used in the underlying routing layer. We use the algorithms described in Section 4.7.1, denoted as *GSP*, *DSP*, *Grd*, *GrdRnd1*, *GrdRnd2*.

Multicast size: size of multicast requests (incl. source node). We use 2 (unicast), as well as 4, 8, and 16 (true multicast requests), which we denote as *M2*, *M4*, *M8*, and *M16*, respectively.

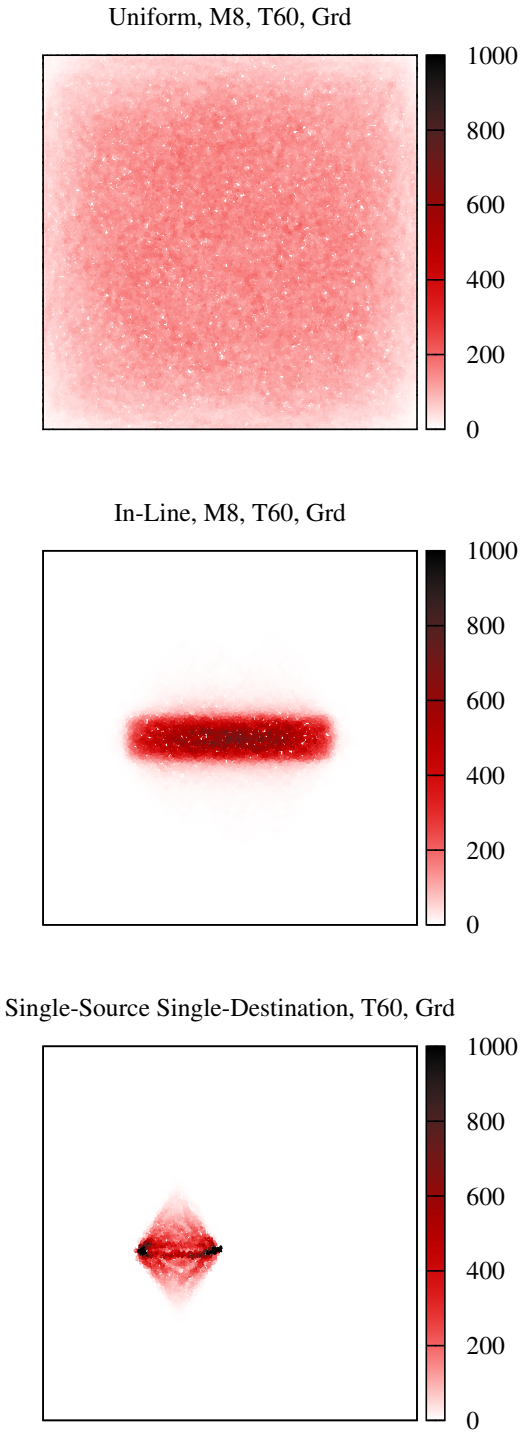

Figure 4.3. Examples of the Three Classes of Workloads

Workload class: location of sources and targets in multicast requests, chosen according to the *uniform* and *in-line* model.

4.7.3 Results

We now report the most important results of the simulation analysis. We first focus on the performance of the underlying geographic routing layer. We found that in all our experiments, the greedy algorithm yields the best results in terms of congestion. As an example, Figure 4.4 shows the network conges-

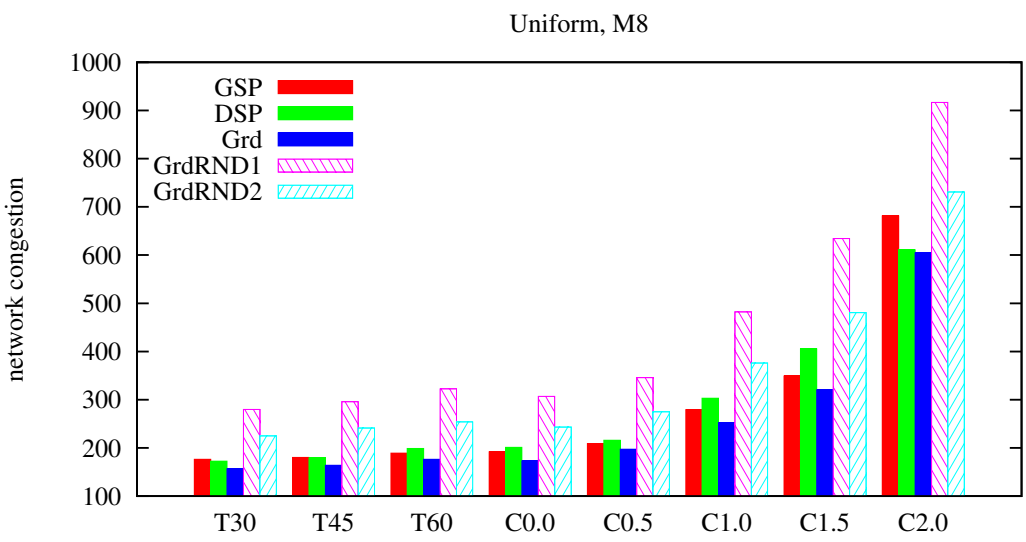

Figure 4.4. Comparison of Geographic Routing Algorithms

tion incurred by the various geographic routing primitives under a workload of uniform multicast requests of size 8, in combination with every variant of our scheme. In these scenarios, the greedy algorithm (*Grd*) is always the one that causes the lowest congestion, and as it turns out, all other scenarios show similar results. This result is particularly interesting and positive because *Grd* is also the simplest geographic algorithm available. Therefore, we dismiss all other underlying routing algorithms for the rest of our analysis.

The next question we consider is how the network congestion is affected by the selection of the intermediate point. The histogram of Figure 4.4 already indicates that the angle-based selection methods *T30*, *T45*, and *T60* work better than the method based on the circular arc for distance factor $\gamma > 1$.

Figure 4.5 confirms this result. The two graphs show the network congestion incurred by the various selection methods as a function of the size of the

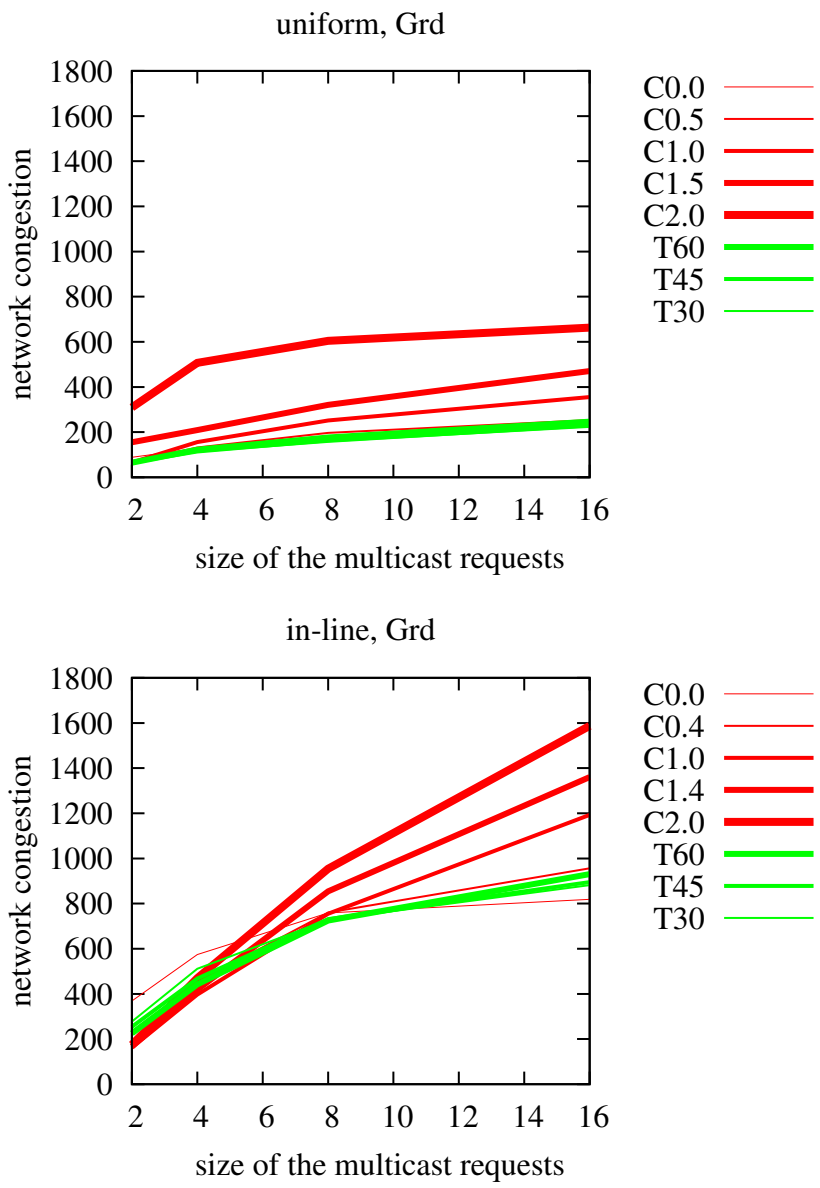

Figure 4.5. Comparison of Intermediate Point Selection Methods

multicast requests, for *uniform* and *in-line* workloads, respectively. The conclusion we can draw from these experiments is that the angle-based schemes achieve the best results, only slightly better than the circular-arc method with radius less than the distance ($\gamma < 1$), and that the circular-arc method shows definitely worse performance with higher radii ($\gamma > 1$) with proportionally worse outcomes in the case of uniform workloads. Also note that for the *in-line* model, the deterministic *C0* algorithm is the worst one for small multicast requests and it becomes the best algorithm for large multicast requests. For small requests, when routing deterministically, in the *in-line* scenario many routing paths overlap and by routing around, the congestion can be reduced. For large requests with all destinations on a line, the message has to be sent along the whole line anyway, so that sending it directly along the line becomes cheaper.

The graphs of Figure 4.5 also demonstrate that our multicast routing scheme performs well in an absolute sense and in particular they seem to indicate that the scheme scales gracefully with a sublinear relation between the size of the multicast requests and congestion. Recall that all workloads consist of 1000 requests, so, for example, in the case of requests of size 16, that means that each of the 1000 messages must be delivered to 15 destinations. Consider this scenario in the extreme case of requests in which all destinations lay on a line (or a narrow band) in the network, which corresponds to the case of the *in-line* workloads. It is interesting to notice that in this case, the scheme is capable of routing all requests in such a way that the maximally-loaded node sees the equivalent of a worst-case set of *unicast* requests.

Chapter 5

Conclusion and Future Work

The Internet for sure has come a long way from its early inception. The Internet of Things World Forum (IoTWF) in May 2017 predicts that the worldwide installed base of Internet of Things (IoT) endpoints will grow from 14.9 billion at the end of 2016 to more than 82 billion in 2025.[1] In addition to this staggering growth, the way we use the Internet has also changed drastically. The Internet was designed to connect and access hosts. In today's Internet, communication is centered on information, which is almost completely decoupled from physical nodes. In other words, consumers are no longer interested in connecting to a particular host, and instead are interested in accessing the data, which must simply be available and authentic.

Information-centric networking (ICN) is a novel, general network architecture intended to better support the more modern information-centric usage of the Internet. The now substantial scholarly literature and the numerous research and engineering projects in ICN cover a variety of different aspects of ICN, ranging from security to in-network caching. And yet, the problem of routing based on true information-centric addresses has remained mostly untouched. In this thesis we proposed a new architecture for ICN, one based on true information-centric addresses, and we developed routing and forwarding algorithm for this architecture.

In particular, we proposed a multi-tree routing scheme with a new tag-based addressing model to describe content. We evaluated this routing scheme for both intra and inter-AS topologies using a realistic application workload. After studying the efficiency of this routing scheme, we presented the design and implementation of TagMatch, an efficient subset matching engine that

[1] IDC Worldwide Internet of Things Installed Base by Connectivity Forecast, March 2017. Link:https://www.idc.com/getdoc.jsp?containerId=US42331917

exploits a hybrid system of CPUs and GPUs. We used TagMatch as the fast forwarding engine for our proposed ICN model. We designed TagMatch as a general subset matching engine and studied its usage in other context such as an advanced, Twitter-like application (with a richer subscription model than the basic "follower" system of Twitter). TagMatch targets applications that perform subset matching between a high-rate stream of queries, each consisting of a relatively small set of tags, with a large database of hundreds of millions of tag sets.

We presented an extensive evaluation of TagMatch in which we test its absolute performance with various workloads and we compare it with the MongoDB database system, with a message forwarding system, and with a system based on a prefix tree that is representative of the most efficient solutions for subset matching we know of. TagMatch outperforms these systems, in most cases with at least an order of magnitude higher throughput. Remarkably, TagMatch can process about five times the average message traffic of Twitter on a single commodity machine, while offering a refined service that dispatches tweets based on the interests of the users rather than only on the publisher of the tweets.

From the technical view point, and also more generally, TagMatch demonstrates the synergistic use of two different hardware architectures, each with its own advantages and drawbacks, to achieve high-throughput information classification. In particular, we designed TagMatch as a pipeline capable of fully exploiting both CPUs and GPUs.

One possible direction for future work can be to include the integration of TagMatch within a full fledged data processing or messaging systems, to measure the benefits that it can bring to such application domains. Another possible future work would be to design another CPU-GPU pipeline for the *consolidate* function. This can improve the performance of partitioning algorithm even more. Such improvement in the speed of *consolidate* function would be beneficial in applications in which the database of sets changes more frequently. Moreover, in studying the effects of GPU pre-filtering, we showed that the performance of TagMatch is blocked by the *key lookup* stage on the CPU side of the pipeline. One possible direction for improvement would be to investigate how we can do this stage more efficiently. Doing so would enable TagMatch to achieve even higher throughput.

In the same area of forwarding, but in a completely different direction, we could investigate other mixed-hardware architectures, where the task of matching and filtering data packets is split over a set of different parts and layers, each with specific hardware characteristics and flow intensities. For example, we imagine that the matching and filtering of information that is

ultimately necessary and useful to applications could be effectively performed in stages. Some stages could be assigned to computational components inside the network, from the core to the edge, or in a data center from the global interconnect to the top-of-rack switch. Other stages could be assigned to the operating system and then ultimately to the application itself.

A promising technology that would support this kind of flexible division of labor between network and application is programmable data-plane, as represented primarily by the P4 language. The design of P4 implies parallel processing over different packets, coupled with pipe-line processing for each packet. This is in contrast with the data parallelism of GPUs that we exploit in our hybrid system of CPUs and GPUs. Still, the high-level architecture of TagMatch is a classic multi-stage pipeline that could be adapted to other mixed-hardware systems including networks with a programmable data plane.

Bibliography

[1] I. Abraham, I. Abraham, C. Gavoille, C. Gavoille, D. Malkhi, and D. Malkhi. Routing with improved communication-space trade-off. In *In 18 th International Symposium on Distributed Computing (DISC*, pages 305–319. Springer, 2004.

[2] I. Abraham, D. Dolev, and D. Malkhi. LLS: A locality aware location service for mobile ad hoc networks. In *Proc. 2nd Workshop on Foundations of Mobile Comp. (DIALM-POMC)*, pages 75–84, 2004.

[3] I. Abraham, C. Gavoille, and D. Malkhi. On space-stretch trade-offs: lower bounds. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '06, pages 207–216, New York, NY, USA, 2006. ACM.

[4] I. Abraham, C. Gavoille, D. Malkhi, N. Nisan, and M. Thorup. Compact name-independent routing with minimum stretch. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 20–24, New York, NY, USA, 2004. ACM.

[5] I. Abraham, C. Gavoille, and D. Ratajczak. Compact multicast routing. In *Proc. of 23rd Symp. on Distributed Computing (DISC)*, pages 364–378, 2009.

[6] I. Abraham, D. Malkhi, and D. Ratajczak. Compact multicast routing. In *DISC'09: Proceedings of the 23rd international conference on Distributed computing*, pages 364–378, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A survey of information-centric networking. *IEEE Communications Magazine*, 50(7):26–36, 2012.

[8] P. S. Almeida, C. Baquero, and A. Cunha. Fast distributed computation of distances in networks. In *CDC*, 2012.

[9] M. Arias, L. J. Cowen, K. A. Laing, R. Rajaraman, and O. Taka. Compact routing with name independence. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '03, pages 184–192, New York, NY, USA, 2003. ACM.

[10] S. Arora. Polynomial time approximation scheme for Euclidean TSP and other geometric problems. In *Proc. 37th Symp. on Found. of Comp. Sc. (FOCS)*, pages 2–11, 1996.

[11] B. Awerbuch, A. B. Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *J. Algorithms*, 11:307–341, September 1990.

[12] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 503–513 vol.2, Washington, DC, USA, 1990. IEEE Computer Society.

[13] L. Barrière, P. Fraigniaud, L. Narayanan, and J. Opatrny. Robust position-based routing in wireless ad hoc networks with irregular transmission ranges. *Wireless Communication and Mobile Computing*, 3:141–153, 2003.

[14] Y. Bartal and S. Leonardi. On-line routing in all-optical networks. *Theor. Comp. Sc.*, 221(1-2):19–39, 1999.

[15] A. Borodin, R. Ostrovsky, and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 312–321, New York, NY, USA, 1999. ACM.

[16] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *Proc. Discrete Algorithms and Methods for Mobility (DIALM)*, pages 48–55, 1999.

[17] P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis. Set containment join revisited. *Knowledge and Information Systems*, pages 1–28, 2015.

[18] J. Broch, D. Maltz, D. Johnson, Y.-C. Hu, and J.Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Mobile Computing and Networking*, pages 85–97, 1998.

[19] C. Busch, M. Magdon-Ismail, and J. Xi. Oblivious routing on geometric networks. In *Proc. 17th Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 316–324, 2005.

[20] C. Busch, M. Magdon-Ismail, and J. Xi. Optimal oblivious path selection on the mesh. In *Proc. 19th Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2005.

[21] A. Carzaniga, K. Khazaei, and F. Kuhn. Oblivious low-congestion multicast routing in wireless networks. In *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '12, pages 155–164, New York, NY, USA, 2012. ACM.

[22] A. Carzaniga, K. Khazaei, M. Papalini, and A. L. Wolf. Is information-centric multi-tree routing feasible? In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, ICN '13, pages 3–8, Aug. 2013.

[23] A. Carzaniga, M. Papalini, and A. L. Wolf. Content-based publish/subscribe networking and information-centric networking. In *Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking*, Aug. 2011.

[24] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, Mar. 2004.

[25] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In A. Feldmann, M. Zitterbart, J. Crowcroft, and D. Wetherall, editors, *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany*, pages 163–174. ACM, 2003.

[26] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the International Conference on Management of Data*, SIGMOD '98, pages 355–366. ACM, 1998.

[27] M. Charikar, P. Indyk, and R. Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming (ICALP 2002)*, 2002.

[28] L. J. Cowen. Compact routing with minimum stretch. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 255–260, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[29] R. Flury and R. Wattenhofer. MLS: An efficient location service for mobile ad hoc networks. In *Proc. 7th Symp. on Mobile Ad Hoc Networking and Computing (MOBIHOC)*, pages 226–237, 2006.

[30] P. Fraigniaud and C. Gavoille. Routing in trees. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, ICALP '01, pages 757–772, London, UK, 2001. Springer-Verlag.

[31] J. Gao and L. Zhang. Trade-offs between stretch factor and load-balancing ratio in routing on growth-restricted graphs. *IEEE Trans. on Parallel and Distributed Systems*, 20(2):171–179, 2009.

[32] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox. Information-centric networking: Seeing the forest for the trees. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, Nov. 2011.

[33] M. Gitter and D. R. Cheriton. An architecture for content routing support in the Internet. In *3rd USENIX Symposium on Internet Technologies and Systems*, Mar. 2001.

[34] A. Goel and P. Gupta. Small subset queries and bloom filters using ternary associative memories, with applications. *SIGMETRICS Perform. Eval. Rev.*, 38(1):143–154, June 2010.

[35] R. Goldman and J. Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 285–296, New York, NY, USA, 2000. ACM.

[36] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, Feb. 1999.

[37] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB '97, pages 386–395. Morgan Kaufmann Publishers Inc., 1997.

[38] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261, Oct. 2003.

[39] L. Hong, G. Convertino, and E. H. Chi. Language matters in twitter: A large scale study. In *ICWSM*, 2011.

[40] A. K. M. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang. NLSR: Named-data link state routing protocol. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-centric Networking*, Aug. 2013.

[41] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[42] R. Jampani and V. Pudi. Using prefix-trees for efficiently computing set joins. In *Proceedings of theInternational Conference on Database Systems for Advanced Applications*, DASFAA '05, pages 761–772. Springer-Verlag, 2005.

[43] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[44] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander. LIPSIN: Line speed publish/subscribe inter-networking. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, Aug. 2009.

[45] B. Karp and H. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proc. 6th Int. Conf. on Mobile Computing and Networking (MOBICOM)*, pages 243–254, 2000.

[46] P. Kolman and C. Scheideler. Improved bounds for the unsplittable flow problem. *J. of Algorithms*, 61(1):20–44, 2006.

[47] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. *SIGCOMM Computer Communications Review*, 37(4):181–192, Aug. 2007.

[48] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proc. 11th Canadian Conference on Computational Geometry*, pages 51–54, 1999.

[49] D. Krioukov, k. c. claffy, K. Fall, and A. Brady. On compact routing for the internet. *SIGCOMM Comput. Commun. Rev.*, 37(3):41–52, July 2007.

[50] F. Kuhn, R. Wattenhofer, and A. Zollinger. Ad-hoc networks beyond unit disk graphs. *Wireless Networks*, 14(5):715–729, 2008.

[51] F. Kuhn, R. Wattenhofer, and A. Zollinger. An algorithmic approach to geographic routing in ad hoc and sensor networks. *IEEE/ACM Transactions on Networking*, 16:51–62, 2008.

[52] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, 2010.

[53] K. A. Laing. Name-independent compact routing in trees. *Inf. Process. Lett.*, 103:57–60, July 2007.

[54] F. Li and Y. Wang. Circular sailing routing for wireless networks. In *Proc. 27th Int. Conf. on Computer Communications (INFOCOM)*, pages 1346–1354, 2008.

[55] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. A scalable location service for geographic ad-hoc routing. In *Proc. 6th Int. Conf. on Mobile Comp. and Networking (MOBICOM)*, pages 120–130, 2000.

[56] Y. Luo, G. H. L. Fletcher, J. Hidders, and P. D. Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *Proceedings of the International Conference on Data Engineering*, ICDE '15, pages 303–314. IEEE, 2015.

[57] B. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for networks of limited bandwidth. In *Proc. 38th Symp. on Foundations of Comp. Science (FOCS)*, pages 284–293, 1997.

[58] C. Maihofer. A survey of geocast routing protocols. *Communications Surveys & Tutorials*, 6(2):32–42, 2004.

[59] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 157–168, June 2003.

[60] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[61] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems*, 28(1):56–99, 2003.

[62] T. Morzy and M. Zakrzewicz. Group bitmap index: A structure for association rules retrieval. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, KDD '98, pages 284–288, 1998.

[63] F. Papadopoulos, D. Krioukov, M. Boguñá, and A. Vahdat. Greedy forwarding in dynamic scale-free networks embedded in hyperbolic metric spaces. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, Mar. 2010.

[64] M. Papalini. *TagNet: A Scalable Tag-Based Information-Centric Network*. PhD thesis, Università della Svizzera italiana, Oct. 2015.

[65] M. Papalini, A. Carzaniga, K. Khazaei, and A. L. Wolf. Scalable routing for tag-based information-centric networking. In *Proceedings of the 1st International Conference on Information-centric Networking*, ICN'14, pages 17–26, Sept. 2014.

[66] M. Papalini, K. Khazaei, A. Carzaniga, and D. Rogora. High throughput forwarding for icn with descriptors and locators. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ANCS '16, pages 43–54, New York, NY, USA, 2016. ACM.

[67] M. Papalini, K. Khazaei, A. Carzaniga, and A. L. Wolf. Scalable routing for tag-based information-centric networking. Technical Report 2014/01, University of Lugano, Feb. 2014.

[68] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance vector routing. In *Proc. of 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[69] L. Popa, A. Rostamizadeh, R. M. Karp, C. H. Papadimitriou, and I. Stoica. Balancing traffic load in wireless networks with curveball routing. In *Proc. 8th Symp. on Mobile Ad Hoc Networking and Computing (MOBIHOC)*, pages 170–179, 2007.

[70] H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proc. 40th Symp. on Theory of Computing (STOC)*, pages 255–263, 2008.

[71] H. Räcke. Survey on oblivious routing strategies. In *Proc. 5th Conf. on Computability in Europe (CiE)*, pages 419–429, 2009.

[72] R. Rantzau. Processing frequent itemset discovery queries by division and set containment join operators. In *Proceedings of the SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '03, pages 20–27. ACM, 2003.

[73] R. L. Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[74] E. Royer and C. Toh. A review of current routing protocols for ad-hoc mobile wireless networks. In *IEEE Personal Communications*, volume 6, April 1999.

[75] E. M. Royer and C. E. Perkins. Multicast operation of the ad hoc on-demand distance vector routing protocol. In *Proc. 5th Int. Conf. on Mobile Comp. and Networking (MOBICOM)*, pages 207–218, 1999.

[76] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Transactions on Networking*, 12(1), Feb. 2004.

[77] H. Takagi and L. Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *IEEE Transactions on Communications*, 32(3):246–257, 1984.

[78] M. Terrovitis, P. Bouros, P. Vassiliadis, T. Sellis, and N. Mamoulis. Efficient answering of set containment queries for skewed item distributions. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 225–236, New York, NY, USA, 2011. ACM.

[79] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, 2001.

[80] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proc. of 13th Symp. on Theory of computing (STOC)*, pages 263–277, 1981.

[81] U. Varshney. Multicast over wireless networks. *Communications of the ACM*, 45(12):31–37, 2002.

[82] R. Wetzker, C. Zimmermann, and C. Bauckhage. Analyzing social bookmarking systems: A del.icio.us cookbook. In *Mining Social Data (MSoDa) Workshop Proceedings*, pages 26–30. ECAI 2008, July 2008.

[83] J. Xie, R. R. Talpade, A. Mcauley, and M. Liu. Amroute: ad hoc multicast routing protocol. *Mob. Netw. Appl.*, 7(6):429–439, Dec. 2002.

[84] F. Yu and R. H. Katz. Efficient multi-match packet classification with tcam. In *Proceedings of the High Performance Interconnects, 2004. On Proceedings. 12th Annual IEEE Symposium*, HOTI '04, pages 28–34, Washington, DC, USA, 2004. IEEE Computer Society.

[85] X. Yu, X. Ban, W. Zeng, R. Sarkar, X. Gu, and J. Gao. Spherical representation and polyhedron routing for load balancing in wireless sensor networks. In *Proc. 30th Int. Conf. on Computer Communications (INFOCOM)*, pages 621–625, 2011.