ROYAL INSTITUTE
OF TECHNOLOGY

Enabling Internet-Scale Publish/Subscribe In Overlay Networks

FATEMEH RAHIMIAN

Licentiate Thesis in
Electronic and Computer Systems
Stockholm, Sweden 2011

TRITA-ICT/ECS AVH 11:08
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-11/08-SE
ISBN 978-91-7501-137-0

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatesexamen i datalogi Fredag den 3 November 2011 klockan 10.00 i sal D i Forum IT-Universitetet, Kungl Tekniskahögskolan, Isajordsgatan 39, Kista.

Tryck: Universitetsservice US AB

Abstract

As the amount of data in todays Internet is growing larger, users are exposed to too much information, which becomes increasingly more difficult to comprehend. *Publish/subscribe* systems leverage this problem by providing loosely-coupled communications between producers and consumers of data in a network. Data consumers, i.e., *subscribers*, are provided with a *subscription* mechanism, to express their interests in a subset of data, in order to be notified only when some data that matches their subscription is generated by the producers, i.e., *publishers*. Most publish/subscribe systems today, are based on the client/server architectural model. However, to provide the publish/-subscribe service in large scale, companies either have to invest huge amount of money for over-provisioning the resources, or are prone to frequent service failures. Peer-to-peer overlay networks are attractive alternative solutions for building Internet-scale publish/subscribe systems. However, scalability comes with a cost: a published message often needs to traverse a large number of uninterested (unsubscribed) nodes before reaching all its subscribers. We refer to this undesirable traffic, as *relay overhead*. Without careful considerations, the relay overhead might sharply increase resource consumption for the relay nodes (in terms of bandwidth transmission cost, CPU, etc) and could ultimately lead to rapid deterioration of the system's performance once the relay nodes start dropping the messages or choose to permanently abandon the system. To mitigate this problem, some solutions use unbounded number of connections per node, while some other limit the expressiveness of the subscription scheme.

In this thesis work, we introduce two systems called *Vitis* and *Vinifera*, for topic-based and content-based publish/subscribe models, respectively. Both these systems are gossip-based and significantly decrease the relay overhead. We utilize novel techniques to cluster together nodes that exhibit similar subscriptions. In the topic-based model, distinct clusters for each topic are constructed, while clusters in the content-based model are fuzzy and do not have explicit boundaries. We augment these clustered overlays by links that facilitate routing in the network. We construct a hybrid system by injecting structure into an otherwise unstructured network. The resulting structures resemble navigable small-world networks, which spans along clusters of nodes that have similar subscriptions. The properties of such overlays make them an ideal platform for efficient data dissemination in large-scale systems. The systems requires only a bounded node degree and as we show, through simulations, they scale well with the number of nodes and subscriptions and remain efficient under highly complex subscription patterns, high publication rates, and even in the presence of failures in the network. We also compare both systems against some state-of-the-art publish/subscribe systems. Our measurements show that both Vitis and Vinifera significantly outperform their counterparts on various subscription and churn scenarios, under both synthetic workloads and real-world traces.

To Amir Hossein

Acknowledgements

I am deeply grateful to Šarūnas Girdzijauskas for his enormous help and support during this thesis work. With his great knowledge and intellect, he not only contributed to every bit of this research, but also taught me how to do research properly, and patiently worked along side me to improve my writing skills. He is an excellent mentor, as well as, a great friend for lifetime.

I am highly privileged to have worked under supervision of Prof. Seif Haridi. His vast amount of knowledge together with his supervision skills made this work possible. Above all, I can never thank him enough for his invaluable understanding and support during the time, when I most needed it.

I am also extremely thankful to Amir H. Payberah for his contributions to this work and the fruitful discussions. He has always supported me not only as a great colleague in research, but also as my beloved husband in life.

I also acknowledge my work at Swedish Institute of Computer Science, SICS. I am thankful to all my colleagues in SICS, particularly in CSL lab and CNS project, who made my work at SICS enjoyable. Specifically, I would like to express my deepest gratitude to Dr. Sverker Janson, who without his support I could never finish this work. He has always been a great source of inspiration to me and has supported me during the most difficult times of my life.

I am also grateful to Prof. Valdimir Vlassov and Dr. Ali Ghodsi for their valuable feedbacks on my work in general, and on this document, in particular.

I also would like to thank my great friends and colleagues at SICS and KTH, Ahmad Al-Shishtawy, Alex Averbuch, Cosmin Arad, Jim Dowling, Laura Feeney, Raul Jimenez, Jöel Höglund, Tallat Mahmoud Shafat, Johan Montelius, Martin Neumann, Flutra Osmani and Roberto Roverso, for all the fruitful discussions and the knowledge they shared with me.

Finally, I would like to thank Fereidoun, Roxane, Zahra and Zohreh, my beloved brother and sisters. I will always be in debt of my family members for their continuous support and endless love.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The amount of data in the digital world surrounding us is increasing very rapidly. According to a study by IBM, “15 petabytes of data are created every day - 8 times the volume housed in all US libraries" [1]. Thus, finding the relevant information is becoming more like looking for a needle in a haystack. Publish/subscribe systems, or pub/sub systems for short, leverage this problem by providing users with only the information they are actually interested in. Users of such systems utilize a subscription service to express their interest in specific data by either subscribing to a priori-known categories of data or defining filters over the content of the information they want to receive. These subscription models are called *topic-based* and *content-based*, respectively [2]. In both models, whenever some new data appears in the system, the interested subscribers are notified.

Publish/subscribe systems are nowadays widely used over the Internet. Web2.0 applications, such as news syndication (RSS feeds), multi-player games, social networks such as Twitter or Facebook, media streaming applications, e.g., Spotify, or IPTV, are a few examples of publish/subscribe systems. Depending on the application, the publish/subscribe service could be bandwidth intensive, as in streaming applications, or time critical, as in stock market applications, or may include a large number of subscriptions, as in Spotify playlists or social networks.

Currently, the majority of these systems use a client/server model and rely on dedicated machines to provide subscribe services. However, with a rapidly growing number of users on the Internet, and a highly increasing number of subscriptions, it is becoming necessary to use decentralized models for providing such a service at a reasonable cost. Moreover, the centralized model raises a privacy problem, since all the user interests are revealed to a central authority, while in the real life most users are reluctant to give away their personal interests for various privacy reasons. Therefore, researchers have turned to peer-to-peer overlays, as an alternative design paradigm to the centralized model. Peer-to-peer overlays, if well implemented,

exploit the resources at the edges of the network to provide a scalable service at a low or almost no cost. The available resources in a peer-to-peer network grow/shrink when more nodes join/leave the system. However, continuous joins and fails in such networks should be gracefully handled in order to provide a reasonable quality of service. Many peer-to-peer publish/subscribe systems have been proposed so far. However, they either

- require a potentially unbounded number of connections per node, which renders the system unscalable, or
- are potentially inefficient in routing, which results in large message delivery latencies, or
- put a heavy and/or unbalanced load on the nodes, which could ultimately lead to rapid deterioration of the system's performance once the nodes start dropping the messages or choose to permanently abandon the system.

1.2 Contributions

The main contributions of this thesis are put together in form of two systems, *Vitis* and *Vinifera*, for topic-based and content-based publish/subscribe models, respectively, that address the shortcomings of the existing systems. These contributions can be summarized as follows:

- introducing novel algorithms for how to construct an overlay that adapts to user subscriptions and exploits the similarity of interests. With the use of gossip, we effectively cluster together the nodes with similar or overlapping subscriptions, while every node maintains only a bounded number of connections. These clusters are later exploited to reduce the amount of traffic overhead that is generated in the network.
- introducing a novel algorithm for leader election inside clusters by using only the undergoing gossiping protocol. These leaders, called *gateways* in Vitis terminology, are utilized to connect clusters of nodes with similar interest, while the generated traffic overhead is kept low.
- building efficient data dissemination paths over the clusters by enabling rendezvous routing over unstructured overlays. This is achieved by injecting structure into an otherwise unstructured overlay, using the ideas in the Kleinberg's model. We guarantee that the event delivery time complexity is in logarithmic order.
- introducing load balancing mechanisms that adapt the overlay structure to the load of the published messages.

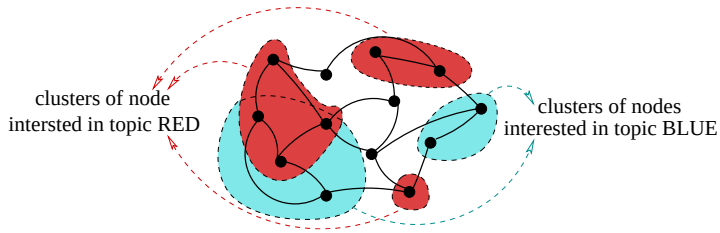


Figure 1.1: The biased neighbor selection puts together nodes with similar subscriptions. Due to bounded node degree, instead of a single cluster per topic, several disjoint clusters are formed. For example, red and blue topics have three and two clusters, respectively.

- combining multiple techniques from various fields, including gossiping, structured overlays and hashing techniques, to construct systems that outperform the existing state-of-the-art solutions.
- implementing and evaluating these systems in simulation, using both synthetically generated and real-world data traces.

In the following sections we will briefly describe our approaches in these systems, and in Chapters 3 and 4 we will go through more technical details of each system, separately.

1.2.1 Vitis: A Topic-based Publish/Subscribe System

Vitis [3] is a topic-based publish/subscribe solution that addresses the shortcomings of the existing system. It requires only a bounded node degree, but generates very low traffic overhead, compares to its counterparts. We use a novel technique for overlay construction, in which nodes exploit the subscription similarities and select as neighbors, nodes with whom they share the most topics. We denote a *cluster* for a topic as a maximal connected subgraph of the overlay, which includes a set of nodes that are all interested in that topic. Due to the bounded node degree requirement, there is no guarantee that all the nodes, which are interested in a topic, connect together. In fact, any number of *clusters* for the same topic can emerge in different parts of the overlay (Figure 1.1).

Although nodes inside a cluster are reachable from one another, in order to make sure a published event for a topic is delivered to all the subscribers, all the clusters of that topic must be also linked together. The path that connects different clusters of the same topic is called *relay path*. Such a path includes nodes that are not interested in the topic themselves. We refer to these nodes as *relay nodes*, hereafter. The challenge is to decrease the number of required relay nodes, while making sure that all the clusters associated with a topic (and therefore, all the nodes interested

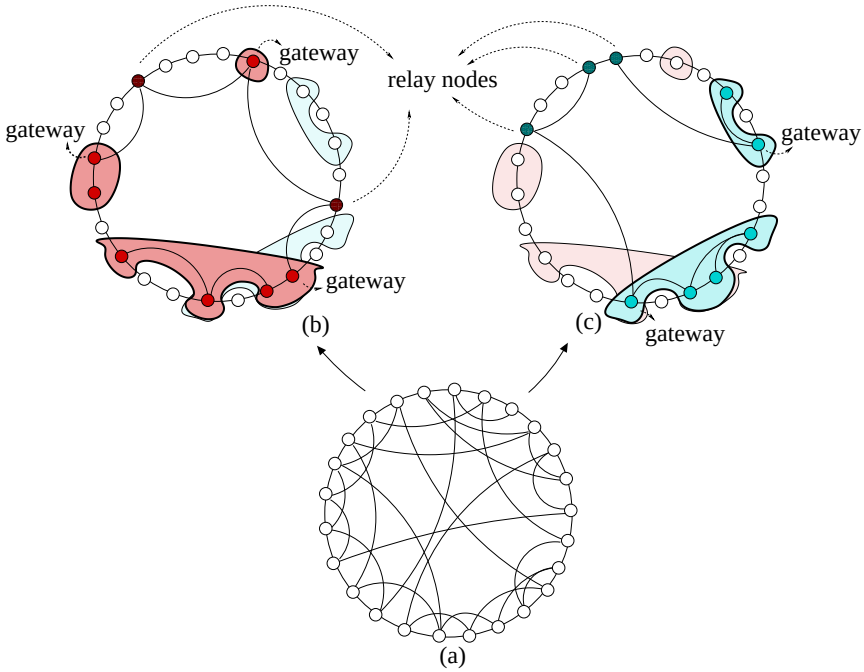


Figure 1.2: The resulting overlay is a single navigable small-world overlay (a), through which disjoint clusters of a topics connect together (b or c). The navigable small-world overlay enables relaying through the nodes, which are not subscribed to the topic, themselves.

in that topic) are linked together. To enable relaying between the clusters, we introduce a novel technique for rendezvous routing [4] on top of an unstructured overlay. For that, Vitis nodes form a navigable small-world overlay (Figure 1.2), which is shown to have the best decentralized routing performance [5, 6].

Moreover, Vitis nodes utilize a novel algorithm to select a number of representative nodes, as *gateways*, in each cluster. The number of gateways for a cluster is proportional to the diameter of the subgraph that represents the cluster. Gateway nodes are responsible for employing the navigable small-world overlay to connect to other clusters for the same topic. They perform a greedy lookup for the topic id, and all meet at the same node, i.e., rendezvous node. This approach is comparable to Scribe or Bayeux, but the difference is that nodes are efficiently grouped together in advance, and instead of each node independently performing the rendezvous routing, only few nodes, i.e., gateway nodes, establish the relay paths. In section 3.3 we elaborate on how the gateway nodes are selected and how the relay paths are established. We also show that the event propagation delay, in terms of

the number of hops, is bounded to $O(\log^2 N)$, in our system. The resulting structure resembles a grapevine, with clusters of grape hanging from the canes, thus, inspired the name Vitis.

We evaluated the performance of Vitis through extensive large-scale simulations, with synthetic data as well as real-world subscription traces from Twitter [7], and churn traces from Skype [8]. We compare our system, with two base-line solutions: (i) a rendezvous routing system which is based on a structured overlay, with a bounded node degree, and oblivious to node subscriptions, and (ii) an unstructured solution that exploits the subscription correlation between nodes, without any bound on node degree. The results show that the traffic overhead in Vitis is between 40% to 75% less than the first base-line solution. We also show that, with a bounded node degree, Vitis always deliver the events to all the subscribers, while the hit ratio degrades in the second base-line solution, when the node degree can not grow indefinitely.

1.2.2 Vinifera: A Content-based Publish/Subscribe System

Vinifera is a solution for content-based publish/subscribe over a peer-to-peer network, which addresses the shortcomings of the aforementioned systems. Similar to Vitis, a key design characteristic of Vinifera is that any node maintains only a bounded number of connections to other nodes, regardless of the number of existing nodes and subscriptions. Vinifera clusters the nodes with similar subscription, but unlike Vitis, clusters in Vinifera are fuzzy, meaning that they do not have a distinct boundary. Note that, subscriptions in Vinifera allow for ranges over different attributes, thus, subscriptions may overlap but they are hardly exactly equal. Nevertheless, Vinifera utilizes a novel technique to put together nodes that can be most helpful to one another, when it come to routing the events. To our knowledge, Vinifera is the first system that exploits subscription similarities in a content-based scheme, to create efficient data dissemination structures.

Moreover, a navigable small-world structure is embedded into the overlay after each node is assigned an identifier, selected from a globally known identifier space. This enables a distributed greedy distance minimizing routing algorithm to find short paths between any two nodes [6], which in turn, allows us to utilize a *rendezvous routing* technique [4], to build efficient data dissemination structures , used by many publish/subscribe systems. However, in contrast to other content-based publish/subscribe systems, e.g., Ferry, that assign a single rendezvous node to handle all the data items of a given attribute (Figure 1.3), Vinifera distributes this responsibility across all the nodes in the system (Figure 1.4).

Vinifera maps the subscription space of each attribute to the same 1-dimensional identifier space that is used for generating node identifiers. This mapping is done using an *Order Preserving Hash Function* [9] [10], or OPHF for short, such that (i) the entire attribute space is mapped to the entire identifier space, and (ii) the order of the values in the attribute space is preserved in the identifier space.

Consequently, every node is responsible for a part of the attribute space, which is

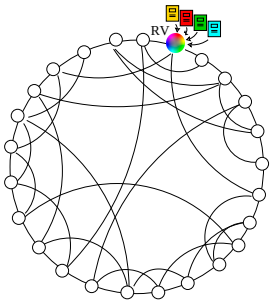


Figure 1.3: Ferry: a single RV node for each attribute is responsible for delivering the published events to the matching subscribers. All the events, regardless of their content, are delivered to the RV node, and are matched against all the registered subscriptions.

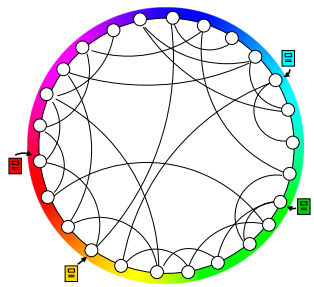


Figure 1.4: Vinifera: the attribute space is mapped to the identifier space and each node takes the responsibility, i.e., becomes the RV node, for a part of the range. The events are then delivered to the corresponding RV nodes, based on their content.

mapped to the range between itself and its predecessor in the identifier space. Moreover, data items which are close to each other in the attribute space are handled by one or few nodes located in the same vicinity of the identifier space. Figure 1.4 depicts an example with one attribute, where different colors represent different hashed values for the attribute. As we will explain in Chapter 4, in case we have more attributes, they are also mapped to the same identifier space. Such a mapping allows us to access the rendezvous nodes responsible for contiguous data ranges of an attribute by performing a range query. To do this, we utilize a showering algorithm technique over the given range [11], which is previously proven to be efficient for the corresponding task. In this approach, the message that contains the range is greedily forwarded towards the peers with identifiers that fall within that range, using several parallel paths, if necessary. Hence, we are able to quickly create dissemination paths (trees) between the subscriber nodes (leaves of the trees) issuing subscriptions over the ranges of a given attribute and the RV nodes (roots of the trees) responsible for the data items falling under these subscription ranges.

Figure 1.5 illustrates how such a tree is built for a node that subscribes to a specific range, e.g., range $[m, n]$. As the trees are constructed from the subscribers to the rendezvous nodes, the subscriptions are registered in all the intermediary nodes of these trees. Later, as shown in Figure 1.6, events are delivered along the reverse paths from the RV nodes to the subscribers. As opposed to the central matching process in systems like Ferry, the matching process in Vinifera is accomplished along the delivery path, i.e., events are partially matched to the registered subscriptions at each step as they are forwarded on the tree. Hence, the load of matching events against subscriptions is distributed across the nodes.

Furthermore, we use an aggregation technique to merge different subscription requests that are received by the nodes. In section 4.3 we will explain, in details, how the aggregation helps in reducing the maintenance cost and facilitates the

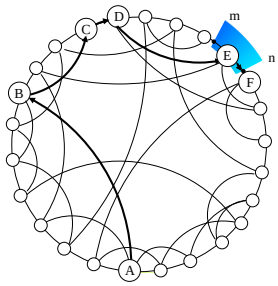


Figure 1.5: Node A subscribes to range [m, n], by sending a lookup for this range. Two consecutive nodes, E, and F, are responsible for this range. All the nodes along the path, including B, C, and D, register the subscription request.

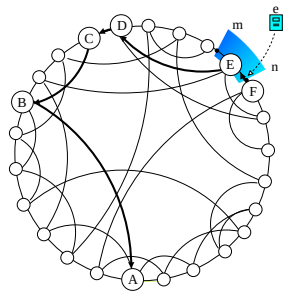


Figure 1.6: An event e in the range [m, n] is published. The corresponding rendezvous node, F, receives the event and forwards it along the reverse subscription path to the subscriber A.

matching process.

Vinifera is also empowered by a load balancing technique that enables it to deal with non-uniform hash functions. Thus, it ensures an evenly distributed load, even in the presence of non-uniform user subscriptions, i.e., when some regions in the identifier space are more popular than the others. The resulting balanced load in Vinifera is of critical importance, not only because it implies fairness and a higher resource utilization, but also, and most importantly, because it enables the system to tolerate churn and massive event publications.

We run extensive simulations to evaluate multiple aspects of the performance of Vinifera, namely scalability, fault tolerance, load balancing and congestion control. We also compare Vinifera to a state-of-the-art solution, equivalent to Ferry [12], and show that, Vinifera decreases the traffic overhead of the network by a factor of three, while the load is evenly distributed among the nodes. Also, the events are delivered to the subscribers up to four times faster. Moreover, Vinifera performs significantly better in the presence of failures in the network, as well as, under high load.

1.3 Delimitations

- **Durability.** *Durability* refers to the property that a generated data item will survive in the system permanently. This property is of great importance for many applications, specially database systems. A publish/subscribe system can also be augmented by a durable storage system, which guarantees the persistency of events, as well as subscriptions. A lot of research is going on to design distributed storage systems and key-value stores which provide such guarantees. However, these works are orthogonal to our work and are considered out of the scope of this document.

- **Content filtering and matching techniques.** There are some interesting work on how to filter data content in the overlay networks [13] [14] [15]. However, these works are orthogonal and can be complementary to our solutions. In particular, we can utilize [13] on top of our dissemination trees in order to better filter out the published content.
- **Security attacks and byzantine behaviors.** Although security issues are practically important in real-world systems, the research work to address such issues are also orthogonal to our work and are considered out of the scope of this research. We assume all nodes behave in accordance with the protocols. However, node and link failures are handled in our systems.

1.4 Outline

The rest of this document is organized as follows. Chapter 2 gives the necessary background for better understanding the utilized approaches in the thesis. It also explores the related work. Chapter 3 presents Vitis, our solution for the topic-based subscription model. It goes through technical details, algorithms, and the experimental results. Likewise, Chapter 4 presents the algorithms and technical details for Vinifera, our solution for the content-based subscription model. It also reports the experimental results on how this system performs in different scenarios and under different workloads. Finally, in Chapter 5 we conclude the work and hint on some future research directions.

Chapter 2

Background and Related Work

In this chapter we explore the necessary background for the thesis, as well as the related work. First, we review peer-to-peer networks and their properties. Next, we present the basics of peer sampling services and how we can build a topology using a peer sampling service. Then we elaborate on publish/subscribe systems and give a survey of the related work.

2.1 Peer-to-Peer Overlays

A peer-to-peer (P2P) overlay is an overlay network that exploits the existing resources at the edge of the network. Each node is represented by a peer, and plays the role of both client and server in the overlay network. Peers cooperate to provide a distributed service, without the need for a single or centralized coordinator/server. The resources in such networks increase as more peers join the network. Thus, P2P networks can potentially scale to a large number of participating nodes without having to dedicate powerful machines to provide the service. BitTorrent and Skype are two well-known examples of such networks. In a P2P network peers can join or leave the network continuously and concurrently. This phenomenon is called *churn*. Also network capacities change due to congestion, link failures, etc. Any such system, therefore, must handle churn in order to provide a reasonable quality of service.

Peer-to-peer overlays are mainly categorized into (i) structured, (ii) unstructured, and (iii) hybrid overlays. In a structured overlay, nodes acquire an identifier from a globally known identifier space and are arranged to form a well defined topology. Such overlays should provide navigability, that is every node should be able to route to any other node in few, usually logarithmic, number of steps. This is achieved by utilizing a greedy distance-minimizing lookup service over the topology. Chord [16], Pastry [17], Kademlia [18], Symphony [19], CAN [20], One-hop DHT [21] and Oscar [22–24] are examples of the structured overlays.

One the other hand, unstructured overlays usually do not have a predefined

topology and nodes randomly discover and select each other to link with. Lookup in these overlays usually takes the form of either flooding or random walk. Gnutella [25] and Kaza [26] are two examples of unstructured overlays.

While structured overlays are more efficient in routing, they need to be constantly maintained in the presence of churn in the network. On the other hand, unstructured overlays are very robust and automatically adapt to the changes in the network, though they can not guarantee a bounded routing time. Hybrid overlays exploit the best of the two worlds and are optimized for specific purposes. In such an overlay, some links are chosen with predefined criteria that lead to better routing performance, while some other links are selected randomly or based on other characteristics that are important for the application. In our systems, we construct hybrid overlays that are specially designed for publish/subscribe purpose.

2.2 Small-world Networks

The small-world phenomenon refers to the property that any two individuals in a network are usually connected through a short chain of acquaintances. The existence of such chains have been long studied by researchers in different sciences, ranging from mathematics and physics to sociology and communication networks.

In 2000, Kleinberg [27] argued that there exist two fundamental components to this phenomenon. One is that such short chains are ubiquitous and the other is that individuals are able to find these short chains, using only local information. Kleinberg introduced the notion of *distance* and showed that in a small-world network two nodes are connected not uniformly at random, but with a probability that is inversely proportional to their distance. More precisely, nodes u and v are connected to one another with probability $d(u,v)^{-\alpha}$, where $d(u,v)$ denotes the distance between the two nodes, and α is a structural parameter. Different values for α yields a wide range of small-world networks, from random to regular graphs. However, Kleinberg mathematically proved that a greedy routing algorithms works best only if α is equal to the number of dimensions in the network. In other words, navigation in a r-dimensional small-world network is most efficient only if nodes u and v are connected to each other with probability $d(u,v)^{-r}$.

Many peer-to-peer systems, such as Symphony [19], Oscar [22, 24] and Mercury [28], have already used Kleinberg's ideas to introduce overlay structures that are efficient in routing. We are also inspired by these works, in order to ensure a bounded routing complexity in our overlays.

2.3 Peer Sampling Services

Peer sampling services (PSS) have been widely used in large scale distributed applications, such as information dissemination [29], aggregation [30], and overlay topology management [31–34]. The main purpose of a PSS is to provide the particpating nodes with uniformly random sample of the nodes in the system. Gossiping

Algorithm 1 T-Man - Active Thread

```
1: procedure ExchangeRT ⟨⟩
2:     neighbor ← selectRandomNeighbor()
3:     buffer ← getSampleNodes()          ▷ provided by the peer sampling service
4:     buffer.merge(RT)                   ▷ RT is the local routing table
5:     Send [buffer] to neighbor
6:     Recv newBuffer from neighbor
7:     buffer.merge(newBuffer)
8:     RT ← selectNeighbors(buffer)
9: end procedure
```

algorithms are the most common approach to implementing a PSS [35–41]. In a gossip-based PSS, protocol execution at each node is divided into periodic cycles. In each cycle, every node selects a node from its partial view and exchanges a subset of its partial view with the selected node. Subsequently, both nodes update their partial views. Implementations of a PSS vary based on a number of different policies [36]:

1. *Node selection*: determines how a node selects another node to exchange information with. It can be either randomly (*rand*), or based on the node's age (*tail*).

2. *View propagation*: determines how to exchange views with the selected node. A node can send its view with or without expecting a reply, called *push-pull* and *push*, respectively.

3. *View selection*: determines how a node updates its view after receiving the nodes' descriptors from the other node. A node can either update its view randomly (*blind*), or keep the youngest nodes (*healer*), or replace the subset of nodes sent to the other node with the received descriptors (*swapper*).

In our work, we employ a light-weight peer sampling service, for providing each node with a uniformly random set of existing nodes in the system. Such service allows our systems to work without the need for any global knowledge at any point.

2.4 Topology Management

The overlay topology management is one of the applications that benefits from peer sampling services. In this thesis, we utilize T-man [31], which is a generic protocol for topology construction and management. In T-man, each node, p, periodically exchanges its routing table (RT) with a neighbor, q, chosen uniformly at random among the existing neighbors in the routing table. Node p, then, merges its current routing table with q's routing table, together with a fresh list of the nodes, provided by the underlying peer sampling service (Algorithms 1, lines 2-7). The resulting list becomes the candidate neighbors list for p. Next, p selects

Algorithm 2 T-Man - Passive Thread

```
1: procedure RespondToRTExchange ⟨⟩
2:     Recv buffer from neighbor
3:     newBuffer ← getSampleNodes()
4:     newBuffer.merge(RT)
5:     Send [newBuffer] to neighbor
6:     newBuffer.merge(buffer)
7:     RT ← selectNeighbors(newBuffer)
8: end procedure
```

a number of neighbors among the candidate neighbors and refreshes its current routing table. The same process will take place at node q (Algorithm 2). The core idea of our topology construction is captured in the neighbor selection mechanism, referred to as *selectNeighbors* in Algorithms 1 and 2. Such flexibility in neighbor selection makes it possible to construct any desirable topology, from a single ring or random graph, to any complex topology like torus, etc. In sections 3.2 and 4.2 we will define the selection mechanisms that we have specifically designed for our topic-based and content-based publish/subscribe systems, respectively.

2.5 Publish/Subscribe Systems

The *publish/subscribe* paradigm provides loosely-coupled communications between producers and consumers of data in a network. Data consumers, i.e., *subscribers*, are provided with a *subscription* mechanism, to express their interests in a subset of data, in order to be notified only when some data that matches their subscription is generated by the producers, i.e., *publishers*. In other terms, subscribers are equipped with a means to filter out the data, in which they have no interest. A data item that is produced/consumed is often referred to as an *event*, in publish/-subscribe terminology.

Publish/subscribe systems are mainly classified into *topic-based* and *content-based* models [2]. In a topic-based model, the events are categorized into predefined *topics* or *subjects*. Each user of the system can then publish/subscribe to events that belong to specific topics. This is comparable to the notion of *group* in group communication, where each generated event that is targeted for a group, is multicasted to the members of that group only.

Although topic-based publish/subscribe systems help users to filter out irrelevant events, they offer a limited expressiveness to the users. The content-based model, improves on the topic-based model by introducing a subscription scheme based on the actual content of the events. Therefore, subscribers can define more fine-grained filters over the events, by introducing a number of constraints over the event content. The event scheme is usually defined using some meta-data and includes a number of *attributes*. The constraints can be in form of basic operations, such as $=$, $<$, $>$, or any combination of them, on each attribute.

To have a better understanding of the challenges and the state-of-the-art for

topic-based and content-based publish/subscribe systems, we focus on the related work in each group separately.

2.5.1 Existing Topic-based Publish/Subscribe Systems

The traditional architectures for publish/subscribe systems are the client-server and broker-based models. In systems based on either of these models, the subscriptions are submitted to a server (or broker). Also publishers send their events to this server (or broker), where the events are matched to the user subscriptions and forwarded to the users, accordingly. Solutions such as Siena [42], Gryphon [43], Hermes [44] or Corona [45] are in this category.

A more recent architecture for designing publish/subscribe systems, replaces the client-server or broker-based models with peer-to-peer overlays. This enables Internet-scale applications with many users as well as many topics. The peer-to-peer overlays can be roughly classified into two main categories: structured and unstructured. Solutions such as Scribe [46] and Bayeux [47] are examples of structured overlay networks, while Tera [48], Rappel [49], StAN [50] and SpiderCast [51] fall into the second category, where a gossip-based approach is utilized. There are also solutions, like Quasar [52] or our solution, Vitis, which use gossiping to construct a hybrid of structured and unstructured overlays for event dissemination.

Regardless of how the overlay is constructed, the main challenge is to guarantee that nodes will receive all the events they have subscribed for, while not being overloaded with a large number of connections or excessive overhead. Tera [48], Rappel [49], StAN [50], and SpiderCast [51] construct a separate overlay for each topic. When a node subscribes to a topic, it becomes a member of that topic overlay. Therefore, published events for that topic are only distributed among the subscriber nodes and the traffic overhead is eliminated. However, nodes should join as many overlays as the number of topics they subscribe to. Thus, the node degree and overlay maintenance overhead grow linearly with the number of node subscriptions. This is, however, impractical for Internet-scale applications, when users subscribe to a large number of topics. We address this problem in Vitis, as nodes maintain a bounded number of connections, regardless of the number of their subscriptions.

To mitigate the scalability problem, SpiderCast [51] takes advantage of the similarity of interest between different nodes. The authors of SpiderCast argue that due to user subscription correlations, a single link can connect a node to more than one topic overlay. Thus, the number of required connections per node decreases. Since the user subscriptions are shown to be typically correlated in the real-world traces [53, 54], this idea works nicely with a limited number of node subscriptions. Nevertheless, the performance and scalability of SpiderCast is unknown, when the number of subscriptions is large or when there is churn in the environment. Moreover, any node in SpiderCast needs to have prior knowledge of at least 5% of other nodes in the system. In contrast, Vitis nodes do not need such a linear-scale amount of information about the other nodes in the system, and can

subscribe to unbounded number of topics. In Section 3.6, we compare a SpiderCast-like system with Vitis and show that SpiderCast nodes suffer from maintaining a large number of connections, in order to receive all the events they subscribed for.

There are also solutions that account for scalability by bounding the number of required connections per node, for example Quasar [52], which is a gossip-based solution, or Scribe [46] and Bayeux [47], which are DHT-based. In Quasar [52], each node exchanges with its nearby neighbors, an aggregated form of subscription information of itself and its neighbors a few hops away. Therefore, a gradient of group members for each topic emerges in the overlay. When a node publishes an event, targeted for a group, it sends multiple copies of the event in random directions along the overlay, and the event is probabilistically routed towards the group members. Quasar obviates the need for an overlay structure that encodes group membership information. However, it is inherently a probabilistic design model, even in a static environment. It also incurs high traffic overhead, since it is oblivious to nodes' subscriptions and involves many uninterested nodes in the event dissemination. In Vitis, on the other hand, we reach a full hit ratio, while minimizing the traffic overhead by organizing similar nodes into clusters.

In Scribe [46] or Bayeux [47], nodes are organized into a Distributed Hash Table (Pastry [42] and Tapestry [55], respectively), where each node maintains $O(\log N)$ connections. Then, a spanning tree is built for each topic, with a rendezvous node at the root, which delivers the events to the nodes that join the tree. This approach, however, forces many nodes to relay the events for which they have not subscribed, as they happen to be on the path towards the rendezvous node. Consequently, such systems suffer from a huge amount of traffic overhead. Vitis nodes also have a bounded node degree and form a tree-like structure per topic. However, unlike Scribe or Bayeux, the leaves in these trees are not single nodes, but groups of nodes, which are subscribed for that topic. We show through simulations, that an efficient clustering of nodes with similar interests, results in trees with far less intermediary nodes, and hence, much smaller traffic overhead.

Another solution, Magnet [56, 57], exploits similar ideas of subscription correlation between the nodes, under the bounded node degree assumption. However, Magnet is purely based on a structured overlay and cannot fully capture the correlation between subscriptions, for it is bounded to one dimensional space, where the structured overlay is constructed. Also, Magnet is less robust in volatile environments, such as the Internet. In contrast, Vitis is not restricted to any dimension while capturing the subscription correlation (since clustering is done in an unstructured way) and as we show in our experiments, it is very robust due to the underlying gossip protocol.

Finally, there is recent work for resource location in clouds [58], which can be interpreted as a publish/subscribe system, though with quite clear differences. In [58], nodes query for a resource with certain attributes, and are redirected to a part of the cloud that contains the resources with requested properties. This work also employs a peer sampling service to build a structured and an unstructured overlay. In the unstructured overlay, resources with similar attributes are placed

close to one another. However, [58] does not guarantee, and in fact does not need, that all the nodes with the queried properties are found. Nevertheless, in Vitis, we make sure that all the subscribers are found and informed of the published event. Moreover, [58] is not applicable for event dissemination, for it enforces a significant load on the nodes in the structured overlay.

2.5.2 Existing Content-based Publish/Subscribe Systems

The research on publish/subscribe systems has been of interest for long. Many solutions are already proposed for topic-based publish/subscribe model [3] [56] [51] [59]. Likewise, there exists a number of peer-to-peer solutions for providing the content-based publish/subscribe service. In Meghdoot [60], for example, each node subscription is mapped to a point in a $2d$ dimensional space, where d is the number of attributes/dimensions in the subscription scheme. Then, a CAN [20] overlay is utilized for routing the messages. Although matching events against subscriptions can be nicely done in Meghdoot, the routing is not efficient, due to the inherent inefficiencies in CAN overlay. Moreover, node degree could grow linearly with the number of attributes. The load on the nodes is also very unbalanced, depending on where in the CAN overlay the node is positioned.

Sub-2-Sub [61] takes a completely different approach. It clusters the subscription space into multiple sub-spaces, where each subspace includes all and only the nodes that are subscribed to the whole subspace. From then on, each subspace is treated like a topic in a topic-based model. A ring is constructed over each subspace for disseminating the events inside that subspace. The problems are two fold: firstly, it is difficult to construct the subspaces, if subscriptions are complex. In Hyper [62], which is a non peer-to-peer solution for content-based publish/subscribe, it is proved that solving such a problem is NP-complete. The existence of churn in the peer-to-peer networks makes this problem even more challenging. Secondly, maintaining a ring per subspace means that if a subscription of a node is split into many subspaces, then the node has to join many overlays at the same time. Therefore, the node degree and maintenance cost could grow very large.

Ferry [12] is yet another approach to enable subscriptions over multiple attributes by employing a structured overlay network. Every node hashes the attribute names and sends its subscription to a rendezvous node, which is responsible for one of the generated hash values, preferably to the closest one. All the subscriptions are then maintained at the rendezvous nodes. Upon an event publication, the event is delivered to all the rendezvous nodes and will be routed towards the subscribers, accordingly. The strong point in Ferry is that the node degree is bounded regardless of the number of attributes in the subscription scheme. However, since the nodes subscribe for the hash of the attribute names, the routing structure solely depends on the subscription scheme in the system. For example, if there is only one attribute in the model, then one rendezvous node and one delivery tree will exist. Therefore, the load on the nodes will be extremely unbalanced. The rendezvous node not only receives all the published events in the system, but also has

to match each and every event against all the existing subscriptions, before relaying the received events.

An effort to solve the problems in Ferry is presented in eFerry [63]. The approach is to use different combinations of several attributes, for subscription registration. The proposed mechanisms exhibits loadable properties only for the pub/sub system with extremely large number of attributes, while is still inefficient for the usual systems with one or few attributes.

Another solution, that also requires a bounded node degree, is CAPS [64]. Similar to Ferry, CAPS uses the rendezvous model for subscription installation and event delivery. The main difference is that instead of a single key per attribute, CAPS generates a set of hash values for each subscription, and installs a node subscription in multiple rendezvous nodes in the overlay. The matching is then performed at those rendezvous nodes and events are forwarded along the overlay links from the rendezvous nodes to the subscribers. The problem in CAPS is that a subscription may be translated into too many keys to be installed, and could potentially result in a high traffic in the network. Moreover, the matching is to be performed centrally at the rendezvous nodes and no mechanism for load balancing is proposed.

In contrast to Meghdoot and Sub-2-Sub, Vinifera nodes maintain a bounded number of connections, while very little load is imposed on each node. Similar to Ferry, eFerry and CAPS, a bounded node degree and rendezvous routing technique for subscription installation and event delivery are used in Vinifera. However, instead of hashing the attribute names, Vinifera nodes hash the the attribute values, using an order preserving hash function [9] [10]. This allows us to use a load balancing technique that distributes the matching load and event delivery load uniformly across all the participating nodes.

There are also some related work on how to filter data content in the overlay networks [13] [14] [15]. However, these works are orthogonal and can be complementary to our solution. In particular, we can utilize [13] on top of our dissemination trees in order to better filter out the published content. Likewise, there exist interesting works on delivery guarantees and ordered deliveries, which are again orthogonal to Vinifera. The focus in Vinifera is how to build a topology that exploits user subscriptions to enable efficient data dissemination, in terms of generated network traffic, delivery latency, and maintenance cost.

Chapter 3

Vitis: A Topic-based Pub/Sub System

In this Chapter, we go through the technical details of Vitis and elaborate on how the Vitis overlay is constructed, how the subscriptions are maintained, and how the events are delivered. Moreover, we evaluate the performance of Vitis and compare it against some base-line publish/subscribe systems.

3.1 Preliminaries

At a high level, Vitis borrows ideas from gossip based sampling services [35] (Section 3.2) and rendezvous routing on structured overlays [4]. While benefiting from these ideas, Vitis employs a technique for selecting nodes that share topic interests (Section 3.2.2), and introduces a novel way of constructing a dissemination structure that minimizes the traffic overhead in the network (Section 3.3).

Every Vitis node maintains a bounded-size *routing table (RT)*, which is a partial list of the existing nodes in the system that the node uses for routing the messages. The entries in the routing table are selected either as (i) small-world connections, or (ii) similarity connections based on a preference function. Hereafter, we refer to these two type of connections as *sw-neighbor* and *friends*, respectively. We also use the term *neighbor* to refer to any of the entries in the routing table, either friend or sw-neighbor.

Moreover, each node has a profile, which includes a unique node id, and the id of topics that the node subscribes to. Node ids and topic ids share the same identifier space and are generated by a globally known hash function that generates ids that are uniformly distributed in the identifier space, e.g, SHA-1. The topic id for topic t is denoted by $hash(t)$, hereafter. Subscribing to or unsubscribing from a topic, is done by adding or removing the topic id to/from the profile.

Every node periodically sends its profile to the nodes in its routing table, to inform them of its own subscriptions. This profile message also serves as a heartbeat

Algorithm 3 Join

```
1: procedure Join ⟨⟩
2:     InitProfile()                                   ▷ subscribe to topics
3:     InitRoutingTable()          ▷ get some neighbors from the bootstrap node
4:     start PeerSamplingService()
5:     do every δt                                     ▷ repeat periodically
6:         ExchangeRT()                                          ▷ Algorithm 1
7:         ExchangeProfile()                                     ▷ Algorithm 6
8: end procedure
```

Algorithm 4 Select Neighbors

```
1: procedure SelectNeighbors ⟨buffer⟩
2:     successor ← findSuccessor(buffer)
3:     buffer.remove(successor)
4:     selectedNeighbors.add(successor)
5:     predecessor ← findPredeccessor(buffer)
6:     buffer.remove(predecessor)
7:     selectedNeighbors.add(predecessor)
8:     sw-neighbor ← buffer.select-sw-neighbor(RANDOM-DISTANCE)
9:     buffer.remove(sw-neighbor)
10:    selectedNeighbors.add(sw-neighbor)
11:    for all node in buffer do
12:        utility[node] ← calculateUtility(node, self)
13:    end for
14:    sortedNeighbors ← utility[].sort()
15:    friends ← sortedNeighbors.top(RT-SIZE −3)
16:    selectedNeighbors.add(friends)
17:    return selectedNeighbors
18: end procedure
```

message, and helps the nodes to constantly maintain their routing tables. When a node fails or leaves, its neighbors will stop receiving heartbeat messages and consequently, its entry will be removed from the routing table of its neighbors.

3.2 Overlay Structure

Vitis utilizes a gossip-based peer sampling service to build a hybrid overlay. Any of the existing implementations for this service, e.g., [36] [35] [41] [40], can be used. When a node joins the overlay (Algorithm 3), it contacts a bootstrap node and receives a number of nodes to start communicating with. Then, the node runs the peer sampling service and periodically acquires fresh random samples of the existing nodes. We are also inspired by T-man [31] for the overlay construction and maintenance. As we explained in Chapter 2 T-man provides a generic framework for building any topology. In Vitis we want to build a hybrid overlay that not only captures the similarity of subscriptions between nodes, but also provides us with a platform for efficient routing. The neighbor selection mechanism is Vitis is depicted in Algorithm 4.

As mentioned previously, the routing table of each node includes sw-neighbors and friend links. We define a system parameter k in Vitis, which determines the number of sw-neighbors in the routing table. The lower k is, the higher the upper bound on the routing cost is [19], while nodes are better grouped together and the traffic overhead decreases. That is, there is trade-off between the traffic overhead and the propagation delay, which can be controlled by k. In Section 3.6 we investigate the impacts of this trade-off on the performance of the system.

3.2.1 Sw-neighbor selection

In order to perform rendezvous routing [4], Vitis nodes establish sw-neighbors by utilizing a mechanism similar to Symphony [19]. Similar to Symphony, Vitis constructs a navigable small-world overlay, which guarantees a bounded routing cost that depends on the node degree. It introduces a distance function in the identifier space, where a neighbor for a node is selected with a probability that is inversely proportional to the distance between the two nodes.

The authors in [19] showed that selecting k links according to this probability function, results in a routing cost of the order $O(\frac{1}{k}\log^2 N)$ messages. For example, if one such neighbor is selected (as in Algorithm 4, line 8), the routing time is bounded to $O(\log^2 N)$. Note that, unlike Symphony, in Vitis nodes establish their sw-neighbors via periodic gossiping.

Moreover, our gossip protocol (Algorithms 1 and 2) enables Vitis nodes to form a ring topology in the identifier space. The ring is required for lookup consistency in the overlay, which is, in turn, required for constructing the relay paths (See Section 3.3). Therefore, two entries of the routing tables are always dedicated for maintaining the neighbors on the ring. Each node selects two nodes with the closest id to its own, in the two directions, among the nodes it has learnt about so far, as its predecessor and successor on the ring (Algorithm 4, lines 2 and 6). Although initially the predecessors and successors may not be correctly assigned, T-Man protocol guarantees that through periodic gossiping the ring topology rapidly converges to a correct ring and is constantly maintained, thereafter [31].

3.2.2 Friend selection

The remaining candidate neighbors are ordered by a preference function. A node, then, selects the highest ranked nodes from this list (Algorithm 4, lines 11-15). The preference function takes into account: (i) the interest similarity of the nodes, as well as (ii) the event publication rate for different topics. It can also be extended to account for the underlying network topology and reduce the cost of data transfer in the physical network. The preference function, gives a pair-wise utility value to

the nodes, according to the following function:

$$utility(i,j) = \frac{\sum_{t \in subs(i) \bigcap subs(j)} rate(t)}{\sum_{t \in subs(i) \bigcup subs(j)} rate(t)} \tag{3.1}$$

where $subs(i)$ indicates the set of topics that node i has subscribed to, and $rate(t)$ is the publication rate of topic t.

If the distribution of published events on different topics is uniform, nodes that have bigger interest overlap relative to the total number of their subscriptions, end up as friends. For example, if node p subscribes to topics $\{A, B, C\}$, node q subscribes to $\{C, D\}$, and node r subscribes to $\{C, D, E, F, G, H\}$, then $utility(p, q) = 0.25$, $utility(p, r) = 0.125$, and $utility(q, r) = 0.33$. That means, node p will prefer q to r, although it shares exactly one topic with both of them. Thus, node p less probably gets involved in the event propagation of events on topics $\{E, F, G, H\}$, in which it has no interest. Likewise, nodes q and r prefer to keep r and q in their local views, respectively.

If the publication rate varies for different topics, the interest overlaps are weighted by the publication rates. For example, if the publication rate for topic t goes to zero, i.e., almost no event is published on t, then t is practically ignored in the preference function. On the other hand, nodes will give a high utility to one another, if they are interested in a common topic that has a high rate of events.

3.3 Relay Path Construction

As we explained in Section 3.2, the routing table size is bounded, thus, not all neighbors with utility greater than zero will be selected. As a result, instead of a unique cluster per topic, multiple disjoint clusters can emerge in the overlay. A cluster for topic t, is a maximally connected subgraph of the nodes that are all interested in t. If topic t has n disjoint clusters, these clusters are numbered and denoted as $C_i[t]$, where i is from 1 to n. To ensure that all n clusters of topic t are connected, some other nodes that are not subscribed to t have to get involved.

We define a *rendezvous node* for topic t, as a node with the closest id to $hash(t)$. Since Vitis constructs a small-world overlay, any node is able to route to any other node in the identifier space. To find the rendezvous node, a node performs a lookup on $hash(t)$, and all the nodes on the lookup path become *relay nodes* for t. This path, which we refer to as *relay path*, can include any kinds of links, e.g., friend, sw-neighbor or ring links (Figure 3.1). This is equivalent to the concept explored in Scribe [46] or Bayeux [47], where nodes subscribe on the path towards the rendezvous node and ultimately build a spanning tree.

In order to minimize the number of relay nodes for a topic, instead of letting each subscriber node route to the rendezvous node, as in, e.g., Scribe, nodes inside

Algorithm 5 Update Profile

```
1:  procedure UpdateProfile ⟨⟩
2:      for all topic in profile.subscriptions do
3:          prop ← initProposal(self, self, 0)                    ▷ (GW, parent, hops)
4:          for all neighbor in RT do
5:              if neighbor.isInterested(topic) then
6:                  new ← neighbor.getProposal(topic)
7:                  if neighbor = new.parent OR new.parent ∉ RT then
8:                      currentDis = distance(prop.GW, hash(topic))
9:                      newDis = distance(new.GW, hash(topic))
10:                     if newDis<currentDis AND new.hops+1 < d then
11:                         prop ← (new.GW,neighbor, new.hops+1)
12:                     end if
13:                     if new.GW=prop.GW AND new.hops+1 <prop.hops then
14:                         prop ← (new.GW, neighbor, new.hops+1)
15:                     end if
16:                 end if
17:             end if
18:         end for
19:         profile.subscriptions.update(topic, prop)
20:         if prop.GW = self then
21:             RequestRelay(topic)                    ▷ perform lookup(hash(t))
22:         end if
23:     end for
24: end procedure
```

each cluster select a number of representative nodes, as *gateways*, to establish the relay path.

Algorithm 5 defines the gateway selection process. To select a gateway for cluster $C_i[t]$, each node in $C_i[t]$ initially proposes itself as gateway (Algorithm 5, line 3). This proposal is piggybacked on the node profile that is periodically sent to the neighbors (Algorithm 6). Likewise, the node receives other proposals from its neighbors, and revises its proposal for the next round (Algorithm 5, line 19). To avoid loops, each proposal also includes the node which proposed the gateway. This node is denoted as *parent* in Algorithm 5. Among the proposed gateways, the node selects as gateway the one that has the closest id to $hash(t)$, measured by *distance* function (Algorithm 5, lines 8 and 9). If the selected gateway, e.g., GW in Algorithm 5, is different from the current proposal, the node increases a counter inside the proposal for GW. This counter indicates the distance of the node to the GW, in terms of hop counts. If this distance exceeds a predefined threshold d, the node ignores the proposal (Algorithm 5, line 10). A gateway node, therefore, is responsible for the nodes, which are a maximum of d hops away from it. Consequently, the number of gateways per cluster becomes proportional to the diameter of the cluster, and can be controlled by the distance threshold d. That implies the worst case propagation delay inside a cluster is bounded to d. Hence, the propagation delay in Vitis is $O(\log^2 N + d)$. Nevertheless, d is a constant that does not depend on N and in all the practical scenarios, it can be set to a value less than $\log^2 N$. Therefore, the overall propagation delay is bounded to $O(\log^2 N)$. As our experiments show, in practice this value is much smaller than this upper

bound.

When a node recognizes itself as gateway for topic t (Algorithm 5, line 20), it initiates the relay path construction by performing a lookup on $hash(t)$. Since all the lookups end up at the rendezvous node (the lookup consistency is ensured by the ring), all the clusters of topic t get connected.

It is important to note that nodes do not need to reach consensus on gateways and multiple gateways can be selected for each cluster. This results in establishment of several relay paths from the same cluster and, therefore, more traffic overhead. However, it does not affect the correctness of the solution and is beneficial because: (i) the overlay becomes more robust, in particular to the failure of gateway nodes or relay nodes along the path, and (ii) the propagation delay inside the cluster decreases, since the events are flooded simultaneously in different parts of the cluster.

Should a gateway node fail or disconnect from the cluster (e.g., due to a change of priorities that are enforced by the preference function), its immediate neighbors would detect the failure (after not receiving the heartbeat messages) and stop proposing it as a gateway. Therefore, in the proceeding rounds, those nodes select a different gateway.

3.4 Event Dissemination

Whenever a node publishes an event on a topic, it sends a notification to those neighbors in its routing table, which are interested in that topic, or act as a relay node for the topic. A node that receives a notification, pulls the event from the sender and forwards the notification to all its own interested neighbors. As a result, the notification propagates inside the cluster of the publisher node. When the notification is received by the gateway node, it is forwarded along the relay path. The notification goes up to the rendezvous node and again down the other existing relay paths, if any other cluster for that topic exists. It, then, reaches the gateway node(s) of those clusters, and will be flooded inside those clusters, accordingly. Figure 3.1 shows an example of how a notification is disseminated in the overlay. Node p publishes a new notification on topic t, and sends it to all its neighbors, which are interested in t. When this notification is received by the gateway node, g, it is forwarded on the relay path towards the rendezvous node, i.e., node t. When node t receives the notification, it sends it to the other existing relay path. Consequently, node m is informed and propagates the notification inside its own cluster. The event is pulled from the same path as the notification propagated along.

3.5 Overlay Maintenance

We use a mechanism similar to T-Man [31] and Scribe [46] for maintaining the routing tables and relay paths, respectively. Every time a node sends its profile to its neighbors, it increments the age of those neighbors (Algorithm 6). When

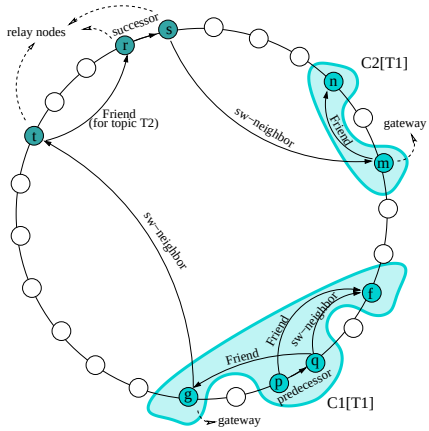


Figure 3.1: Node p publishes a notification inside its own cluster. The notification is flooded inside the cluster. It is also forwarded to the relay node t through the gateway g. The notification moves along the relay path up to the rendezvous node r, and then reaches the other existing clusters. Next, it is flooded inside those clusters.

it receives back a response from the neighbor, it marks that neighbor as fresh, by reseting its age to zero (Algorithm 7, line 3). After a predefined threshold, the stale entries are removed from the routing tables. This threshold determines the failure detection speed. The lower the threshold, the faster the failure detection is. However, if the threshold is too low, then the rate of false positives, due to the congestion in the network and varying link delays, increases. By increasing the threshold, the responsiveness of the failure detection can be traded off for more accuracy.

As we described earlier, the overlay is constructed by gossiping. Through gossiping, clusters are formed, gateway nodes are selected, and relay paths are established. The overlay maintenance is conducted in exactly the same way. When a node leaves the system or modifies its subscriptions, the friend selection mechanism in the proceeding rounds captures this change and routing tables are updated accordingly. If the node is a gateway, then its direct neighbors in the corresponding cluster will notice the change and revise their proposals for selecting a new gateway. If the node is a relay node or rendezvous node, the proceeding lookups by their neighbors on the relay path, will return a substitute node. Consequently, the overlay adapts to the changes in the network, while nodes constantly acquire fresh information through their neighbors.

Algorithm 6 Exchange Profile - Active

```
1:  procedure ExchangeProfile ⟨⟩
2:      profile ← UpdateProfile()
3:      for all neighbor in RT do
4:          if neighbor.age > THRESHOLD then                ▷ remove the stale neighbors
5:              RT.remove(neighbor)
6:          else
7:              RT.neighbor.IncrementAge()
8:              Send [profile] to neighbor
9:          end if
10:     end for
11: end procedure
```

Algorithm 7 Exchange Profile - Reactive

```
1:  procedure RespondToExchangeProfile ⟨⟩
2:      Recv profile from neighbor
3:      RT.update(neighbor, profile, 0)          ▷ 0 indicates the age of this neighbor
4:  end procedure
```

3.6 Experimental Results

We implemented Vitis and two base-line solutions in Peersim [65], a simulator for modeling large scale peer-to-peer networks. The base-line solutions are:

- RVR: a structured RendezVous Routing solution that builds a multicast tree per topic, equivalent to that of Scribe [46] or Bayeux [47], with fixed node degree.
- OPT: an unstructured subscription aware solution that constructs an Overlay Per Topic, while minimizing node degrees by exploiting the subscription correlations, similar to SpiderCast [51].

To make the three systems comparable they use the same peer sampling service (Newscast [40]) and overlay construction protocol (T-Man [31]).

We evaluate Vitis against RVR and OPT with subscription patterns, generated from a synthetic model as well as real-world Twitter traces [7]. We investigate the impact of varying publication rates and routing table sizes on the performance of the systems. Moreover, the robustness of Vitis under churn is evaluated by utilizing traces from Skype [8].

In our simulations, we measure the following metrics:

- *Hit ratio*: The fraction of events, on all topics, that are received by the subscriber nodes;
- *Traffic overhead*: The proportion of relay (uninteresting) traffic that nodes experience;

- *Propagation delay*: The average number of hops events take to reach to all the subscriber nodes.

3.6.1 Experimental settings

We measure the performance of Vitis, RVR, and OPT with 10,000 nodes. Unless otherwise mentioned, k is set to 3, the routing table size is set to 15, d is set to 5, and different topics have the same rate of publication.

We generate three subscription patterns to model different levels of interest correlation. This data generation model was inspired by a work of Wong et al [66]. The subscription patterns are:

- *Random*: nodes select 50 out of 5000 topics uniformly at random;
- *Low correlation*: nodes group 5000 topics into 100 buckets and select 50 topics uniformly at random from 5 different buckets (10 topics from each bucket);
- *High correlation*: nodes group the 5000 topics into 100 buckets and select 50 topics uniformly at random from 2 buckets (25 topics from each bucket).

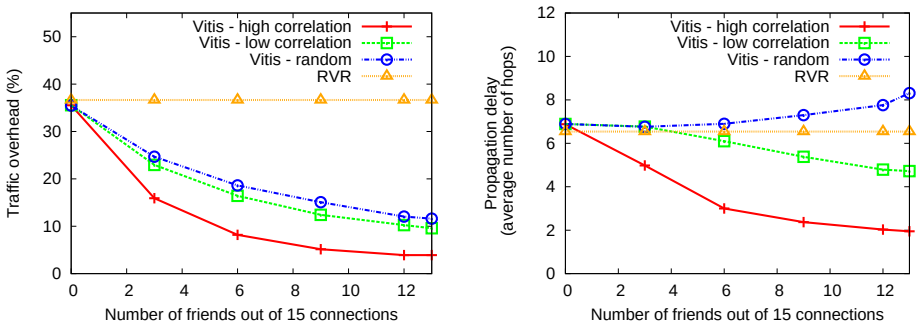
Note that, in all the above subscription patterns, the average topic popularity, i.e., the population of nodes subscribed to a topic, is uniform. Whereas, the distribution of interest correlation, captured by Equation 3.1, is different in the three patterns. Since RVR exhibits similar behavior with random and correlated subscriptions, we draw only a single line for it in the plots. Moreover, since SpiderCast is targeted for real-world scenarios with high subscription correlation, we investigate the performance of OPT only with Twitter subscriptions.

3.6.2 Friends vs. Sw-neighbors

In this experiment, we investigate the performance impact of varying the number of friends versus sw-neighbors. We bound the node degree to 15, that is, each node has a routing table of size 15, among which two links are dedicated for the predecessor and the successor of the node. That means, nodes have at least two sw-neighbors in all the experiments. The rest of the links can be selected, either as a friend or as sw-neighbor.

The results showed that both Vitis and RVR have 100% hit ratio in all settings. As we observe in Figure 3.2a, when more friends are selected, the traffic overhead in Vitis drops significantly. With correlated subscriptions, this traffic reduced by a factor of 88%. Even when the subscriptions are random, the traffic overhead in Vitis is less than one third compared to that of RVR. That shows Vitis is able to exploit even the slightest similarities between nodes subscriptions.

As it is shown in Figure 3.2b, nodes with correlated subscriptions experience a better delivery time as well. The propagation speed improves when more friend links are selected. This is due to the fact that selecting more friends results in



(a) Traffic overhead (b) Propagation delay

Figure 3.2: Measurements with varying number of friends

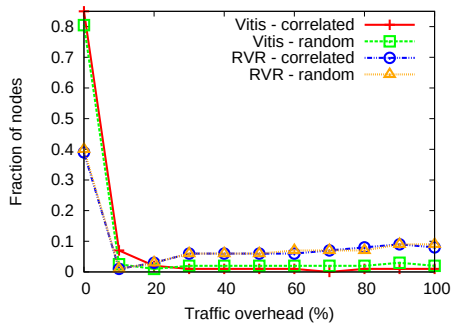
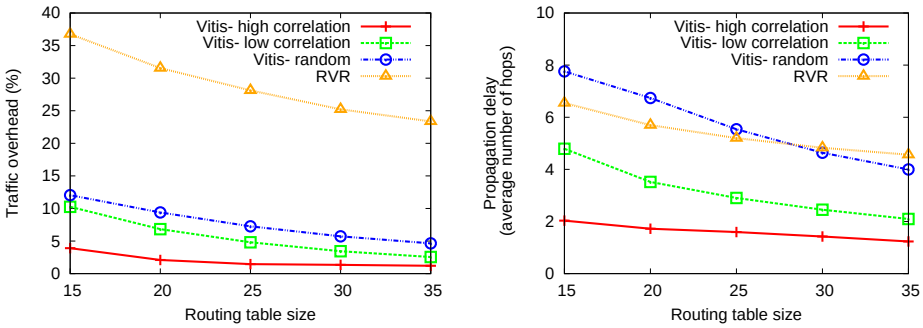


Figure 3.3: Distribution of traffic overhead

a better clustering of nodes with similar subscriptions. Thus, instead of having many small clusters, the overlay moves towards having fewer, but bigger clusters. Since the events are very quickly disseminated inside clusters (by flooding), most of delay is caused by the inter-cluster routing. Therefore with fewer clusters, the event dissemination happens much faster. For random subscriptions, however, the overlay ends up having multiple small clusters per topic. Therefore, inter-cluster routing plays an important role for delivering the events to the subscriber node. Since replacing sw-neighbors with friend links degrades the navigability of system, the improved traffic overhead in this case, comes at the cost of higher propagation delay. However, as discussed in section 3.2, the propagation delay in our system is bounded to $O(\log^2 N)$.

Moreover, one might argue that although the average traffic overhead is reduced in Vitis, a high load is imposed upon gateway nodes, rendezvous nodes, or other relay nodes. Therefore, we show the traffic overhead distribution among the nodes



(a) Traffic overhead (b) Propagation delay

Figure 3.4: Measurements with different routing table sizes

in the overlay. Figure 3.3 shows that while the fraction of nodes with 10% overhead is increased, the fraction of nodes that have an overhead more than 20%, drops to less than one third in Vitis, compared to that of RVR. This shows that Vitis, not only reduces the average traffic overhead, but also improves the distribution of this traffic among the nodes.

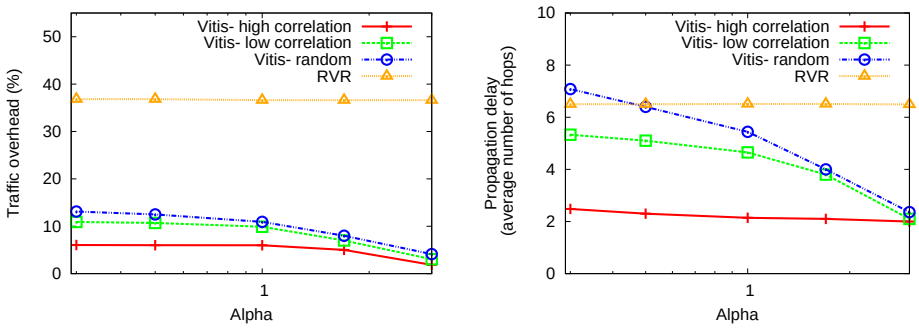
In the rest of our experiments we set one predecessor, one successor, and one sw-neighbor for each node. The rest of the links are selected as friends.

3.6.3 Changing the routing table size

In this experiment we compare the performance of Vitis to RVR, while changing the routing table size from 15 to 35. As it is shown in Figure 3.4a, when nodes maintain bigger routing tables, the traffic overhead, as well as the propagation delay, decreases in both systems, though, for different reasons. In RVR, this improvement is because the rendezvous routing performs better, i.e., in fewer number of hops, with more small-world links. Thus, more efficient spanning trees with less intermediary nodes are constructed. In Vitis, however, the number of sw-neighbors are fixed and the additional entries in the routing tables are used for adding friend links. Therefore, nodes are grouped together more efficiently and fewer relay paths per topic are required. This means inter-cluster routing constitutes a smaller part of the event dissemination. This explains why event delivery latency in Vitis with random subscriptions, outperforms the RVR system, when the routing table size exceeds 30 entries.

3.6.4 Changing the publication rate

So far we have assumed a uniform distribution of published events on each topic. However, the publication rate of topics does not have to be uniform. In fact, usually



(a) Traffic overhead (b) Propagation delay

Figure 3.5: Measurements with different publication rates

there are a few hot topics with a high rate of publications, while other topics have a low publication rate. In this experiment, we show how our solution adapts to different publication rates. We employ a power-law function, with a parameter α, to define the distribution of events rate on different topics. We change α from 0.3 to 3, and evaluate the behavior of Vitis versus RVR. Note that the X-axis in Figure 3.5 is in the log scale. When α is close to 0.3, the distribution is similar to a uniform distribution as in the previous experiments. However, when α increases the distribution becomes more skewed. In the extreme case, when α is 3, almost all the events are published on a single topic.

When the publication rate for different topics becomes more skewed, Equation 3.1 gives a higher utility value to the nodes that are interested in the hot topics. Thus, such topics end up having fewer and better connected clusters. This effect is similar to when the correlation level is increased. That is why in Figure 3.5, the performance of the scenario with random subscriptions gets closer to that of the scenario with high correlation, when α is increased.

Note that, while hot topics are prioritized, topics with less events might experience higher traffic overhead and propagation delay. However, since hot topics constitute most of the published events, and they are propagated efficiently, an overall improvement is achieved.

3.6.5 Real world subscriptions

In this experiment, we evaluate Vitis with both RVR and OPT. We use a subscription pattern extracted from nearly 2.4 million Twitter users [7]. Each node in Twitter plays a dual role, that is, it can follow (subscribe to) other nodes, and it can be followed by others (as a topic). Thus, both topics and nodes refer to the users of the system. We analyzed the available data set and came up with the statistical results reported in Figure 3.7. The distribution of nodes in-degree and

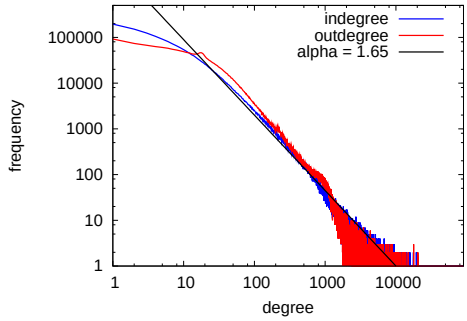


Figure 3.6: Distribution of in-degree and out-degree in Twitter

	In-degree (followers of a node)	Out-degree (subscriptions of a node)
Average	87.8797	87.8778
Variance	965262.3440	171085.2826
Std. Deviation	982.4776	413.6245
Max	349573	172268

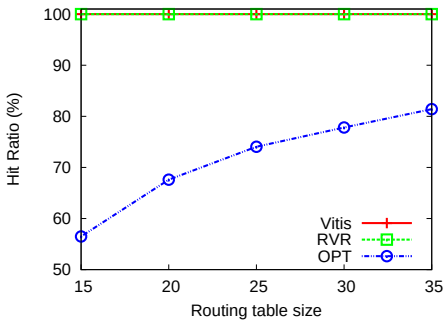
Figure 3.7: Summary of statistical analysis of available Twitter data set

out-degree are modeled by a power-law distribution with an estimated parameter of 1.65 (Figure 3.6).

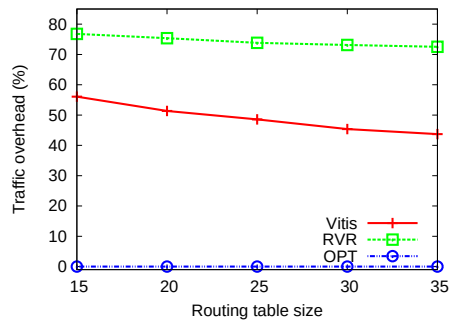
We took a sample of nearly 10000 nodes, by performing multiple breath first searches (BFS) [67]. Initially we randomly selected a number of nodes from the dataset. Then we added to this sample, all the subscriptions of these nodes, i.e., nodes being followed by the selected nodes. Next, we extracted all the relations (following or being followed) between these nodes. Finally, we removed subscriptions to the nodes outside the sample. In order to ensure that this approach preserves the properties of the complete log, we took several samples and the similarity of in-degree and out-degree distribution of the samples and that of the full log was confirmed.

Unlike our previous configurations, in these experiments the number of subscriptions per node is not the same. We changed the routing table size from 15 to 35 and investigated the impact on the hit ratio, traffic overhead and propagation delay. Moreover, we measured the hit ratio for a node, 10 seconds after the node joins the system. That means a node is expected to receive the subscribed-to events, which are published 10 seconds after its joining time.

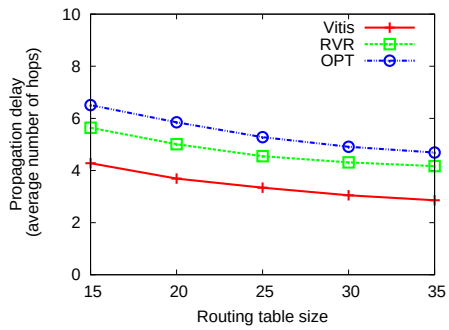
Figure 3.8a shows that the hit ratio in Vitis and RVR is 100%, while OPT with a bounded node degree can not achieve a full hit ratio. Even when the node degree is 35, OPT can only hit 80% of the subscribers, on average. In order to reach a 100% hit ratio, OPT needs to be free of any bound on the node degree. We performed another experiment to investigate the performance of OPT with unbounded node



(a) Hit Ratio



(b) Traffic overhead



(c) Propagation delay

Figure 3.8: Measurements with Twitter subscription patterns

degree, and plotted the node degree distribution in Figure 3.9. As can be seen in this figure, more than two third of the nodes have a degree higher than 15. Also, 0.3% of nodes have a degree higher than 200 (Maximum observed degree is 708), which is not shown in the figure. This implies OPT-like solutions, that only rely on exploiting subscription correlations, can not scale in real world scenarios.

In contrast, OPT outperforms Vitis and RVR with respect to traffic overhead. Since OPT constructs a separate overlay per topic, the events are only disseminated among the subscribers and there is no traffic overhead at all. Figure 3.8b shows the traffic overhead of the three systems. As it is shown, Vitis and RVR has a higher level of overhead compared to Figure 3.4a, which is due to the increased number of subscriptions (on average 80 subscriptions per node). Also, the number of topics in this experiments is doubled, since there are as many topics as the number of nodes. Therefore, the average population of nodes that are interested in a topic is less than the previous experiments. However, even with only 15 links per node, Vitis has 30% less traffic overhead compared to RVR. With 35 links per node, the

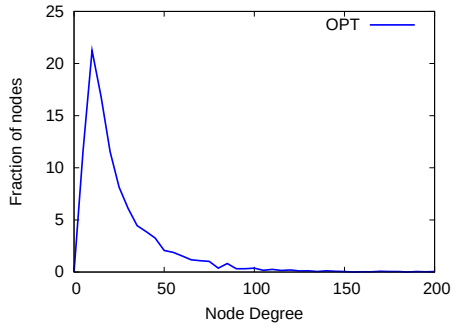


Figure 3.9: Node degree distribution in OPT

traffic overhead in Vitis decreases to 43%, which is 40% better than RVR.

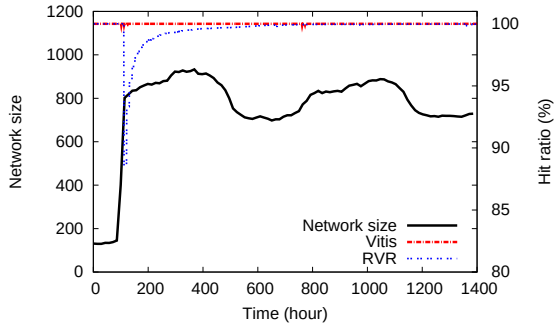
The propagation delay in all three systems exhibits a similar trend when the routing table size increases (Figure 3.8c), while Vitis is more than 1.5 times faster than RVR and 1.7 times faster than OPT. Note that due to the navigable structure, the delay in Vitis and RVR is bounded. However, a topic overlay in OPT might be any arbitrary graph and therefore there is no upper bound on the propagation delay.

3.6.6 Vitis under churn

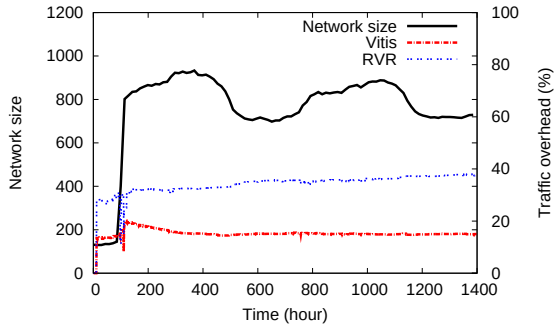
In this experiment, we use a scenario with churn, i.e., a scenario in which nodes can join or leave at any time. We use a real world trace [8], which monitors a set of 4000 nodes participating in the Skype superpeer network for one month beginning September 12, 2005. The routing table size is bounded to 15, and a uniform publication rate for the topics is considered. Like the previous experiment, the hit ratio for a node is calculated 10 seconds after the node joins the system. We compare Vitis with RVR and observe that, due to the underlying gossip mechanism, both solutions react nicely to the churn and adapt to the changes of network.

As Figure 3.10a shows, although both systems can tolerate moderate churn, under *flash crowds*, i.e., a large number of nodes join at nearly the same time, the hit ratio in RVR goes down to 87%. That is because the stabilization time takes longer, and while the structure is not converged to a connected subgraph per topic, nodes may miss some events. This effect is also observable in our system. However, the worst case hit ratio is about 99%. This is because as soon as a node finds a group-mate for a topic, it can receive the corresponding events on that topic, without the need for establishing a relay path independently.

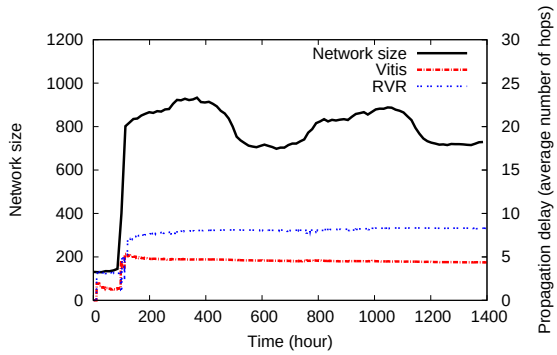
We also observe in Figure 3.10b that the traffic overhead in both systems does not change much over time. However, under flash crowds, the traffic overhead in RVR drops sharply. This is not an advantage though, because the relay paths are not established properly and nodes are missing their desired events (that is why the



(a) Hit ratio



(b) Traffic overhead



(c) Propagation delay

Figure 3.10: Measurements with Skype trace for churn in the network

hit ratio drops as well). In contrast, the traffic overhead in Vitis slightly increases under flash crowds, because nodes inside the groups are not yet informed about their group-mates and therefore several gateway nodes start to build up the relay paths

towards the rendezvous point. After a while, however, when the churn is moderate, the number of gateways and, consequently, the traffic overhead decrease. Likewise, Figure 3.10c shows that the propagation delay does not change in moderate churn. However, the increased level of delay after the flash crowd is due to the bigger size of the network.

Chapter 4

Vinifera: A Content-based Pub/Sub System

In this Chapter, we go through the technical details of Vinifera. We explain the overlay construction mechanism that we used specifically for the content-based model. We elaborate on the subscription installation and maintenance, and event delivery. We also present the experimental results that show the performance of Vinifera and compare it against a state-of-the-art solution.

4.1 Preliminaries

Each Vinifera node has a *profile* that contains a node identifier and the node subscription. The node identifier is selected uniformly at random from a globally known identifier space. The subscription scheme in the system includes n attributes from A_1 to A_n, of any type, where attribute i could take values between $v_{i_{min}}$ and $v_{i_{max}}$. We map this range to the entire node identifier space, by applying an order preserving hash function (OPHF) [9] [10] over the values that are valid for each attribute. An order preserving hash function guarantees that

$$if \quad v > u \quad then \quad OPHF(v) > OPHF(u)$$

For the sake of simplicity, from now on we refer to the hashed value $OPHF(v)$, only as v. Each node subscribes in the system by introducing a number of constraints, denoted as C, over a subset of attributes. A constraint is in form of specifying either an exact value (equality), or a range of values that are valid for the attribute. A subscription S is defined as a conjunction between several constraints. Figure 4.1 shows a system with two attributes A_1 and A_2 and four subscriptions: X, Y, Z and W. Subscriptions are shown by rectangles that specify the ranges over the two attributes. For example, node p with subscription X is modeled as:

$$S_p : C_{A_1} \wedge C_{A_2}$$

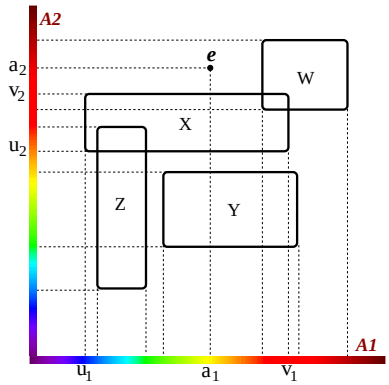


Figure 4.1: A subscription scheme with two attributes A_1 and A_2 and four subscribers. All the attributes are mapped to the same 1-d identifier space. Rectangles X, Y, Z and W depict subscriptions for nodes p, q, r, and t, respectively. Any event, e.g., e, is a point in the attribute space.

$$\begin{cases} C_{A_1} : A_1 \in [u_1, v_1] \\ C_{A_2} : A_2 \in [u_2, v_2] \end{cases}$$

A data item, or *events*, e.g., $e\{a_1, a_2\}$, is a point in the attribute space, with exact values for all the attribute. An event *matches* a subscription, if each and every v_i satisfies the corresponding constraint C_i. For example event e in Figure 4.1 satisfies C_{A_1}, as a_1 lies in the range $[u_1, v_1]$, but not C_{A_2}, so it does not match node p's subscription.

4.2 Overlay Structure

The main building block in Vinifera is a gossip-based peer sampling service [35], similar to Cyclon [41] or NewsCast [40], which is periodically executed by all the nodes. We employ T-Man [31], a topology management protocol, on top of this peer sampling service, to enable nodes to select their neighbors, based on their interest, identifier, and load. The core idea of our topology construction is captured in the neighbor selection mechanism, on which we elaborate. Every node selects three types of links:

- *ring links*,
- *small-world links*, and
- *friend links*.

First of all, each node prefers the two neighbors that have the closest identifiers to it in both directions in the identifier space. When all the nodes connect to such neighbors, a ring topology is shaped up. Therefore, we refer to these two links as

Algorithm 8 Select Primary Attribute

```
1: procedure selectPrimaryAttribute ⟨⟩
2:     for all A_i in self.S do
3:         rank(A_i) ← 0                                   ▷ initialize the rankings
4:     end for
5:     for all n in self.neighbors do
6:         for all A_i in self.S do
7:             if neighbor.S.contains(A_i) then
8:                 selfC_i ← self.S.getC(A_i)              ▷ get the constraint over attribute A_i
9:                 nC_i ← n.S.getC(A_i)
10:                if overlapping(selfC_i, nC_i) ≠ ∅ then
11:                    rank(A_i) =rank(A_i) + 1
                                   ▷ increment the rank for each overlapping interest
12:                end if
13:            end if
14:        end for
15:    end for
16:    A_p ← attribute where rank(attribute) is highest
17: end procedure
```

ring links. The ring topology guarantees the existence of a path between any two nodes, and ensures the lookup consistency in the overlay, which is later required.

To boost the routing efficiency, each node also selects K_{sw} *small-world links*, based on the idea introduced by Kleinberg [5]. More precisely, node p selects node q as a small-world link, with a probability inversely proportional to the distance from p to q in the identifier space. It is proved that having K_{sw} such neighbors enables a poly-logarithmic routing cost in the overlay ($O(\frac{1}{K_{sw}} log^2 N)$) [19].

In addition to the ring links and small-world links, every Vinifera node also selects K_f *friend links*. These links connect together nodes with similar subscriptions. In a system with only one attribute the similarity between two nodes p and q is captured by

$$Utility(p, q) = \frac{S_p \cap S_q}{S_p \cup S_q} \tag{4.1}$$

where, S_i contains the range(s) that node i has subscribed to. As we will explain in Section 4.4, when an event is published, multiple copies of it are propagated in the overlay, one for each attribute. Therefore, to guarantee the event delivery, it is enough if a node is efficiently located in a cluster associated with any of the attributes. In other words, it is enough for a node to cluster with the nodes that are similar to it on only one attribute. The clusterization, i.e., the friend links selection, is completed in two steps:

1. A node first examines the subscriptions of its candidate neighbors to select an attribute, across which it has more neighbors with overlapping ranges. We refer to this attribute as the *primary attribute* for the node. For example, in Figure 4.1 node p, with subscription X, has three neighbors with overlapping subscriptions on attribute A_1 (Nodes q, r, and t) and two neighbors with overlapping subscriptions on attribute A_2 (nodes q and r). So, p selects A_1

as its primary attribute. Algorithm 8 illustrates how the primary attribute is selected. The basic idea is that a node assigns a rank to each of the attributes in its own subscription. The rank of an attribute is calculated by counting the number of neighbors with an overlapping interest on that attribute (Lines 10, 11). Finally, the attribute with the highest rank is selected as the primary attribute (Line 16).

2. Next, the node uses the utility function (Function 4.1) on the primary attribute and biases its neighbor selection towards selecting those nodes with higher rank as its friend links. For example, in Figure 4.1 node p prefers node q with subscription Y over nodes r and t.

The combination of ring links, small-world links and friend links results in a hybrid overlay, on top of which we build the data dissemination paths. We show, in the experiment section, that such a hybrid topology performs significantly better than a pure small-world overlay, such that it not only reduces the unnecessary traffic in the network, but also improves the routing efficiency.

4.3 Routing and Subscription Installation

Vinifera nodes utilize a *rendezvous-based routing* [4] for installing their subscriptions in the overlay. A *rendezvous node* for a value v is a node with the closest, but higher, identifier to v. To find the rendezvous node for value v, a node performs a greedy lookup for v, as shown in Algorithm 9, where the distance to v is minimized at each step. In Vinifera, nodes not only route towards exact values, e.g., v, but also towards ranges of values, e.g., $[u, v]$. To enable range queries, nodes apply an order preserving hash function [9] [10] on the values that are specified in their subscription. Then, a showering algorithm [11] is performed for the hashed values in the range of interest, i.e., when a node receives a lookup request for a range, it forwards the request to all the nodes that fall into the responsibility range. Lines 15 to 17 in Algorithm 10 describes how the showering mechanism is performed. In case the node does not know any node in the requested range (line 18), it just follows a routine greedy routing, i.e., it forwards the request to a node that has a closer, but not higher, identifier to the range than itself. The lookup ends at one or more consecutive rendezvous nodes that are each responsible for a part of the range.

Figure 4.2 illustrates the lookup process. Assume node Q wants to subscribe for the hashed values from 75 to 85. Among its neighbors, it selects node R, which has the closest, but not higher, node identifier to the requested range (Line 19 in Algorithm 10). The request is, therefore, sent from Q to R. Node R forwards the request to its neighbor S, which falls into the requested range (Line 10 in Algorithm 10). Node S takes the responsibility for the sub-range [75, 78], and also forwards a request for the remaining sub-range to node T (Line 6 in Algorithm 10). As a result of such a lookup, a path from the subscriber node to the rendezvous

Algorithm 9 Point Query

```
1:  function findNextHop(value)
2:      nextHop ← self;
3:      for all neighbor in self.neighbors do
4:          if (neighbor.id > nextHop.id ∧ neighbor.id < value) then
5:              nextHop ← neighbor;
6:          else if (nextHop.id > value ∧ neighbor.id > nextHop.id) then
7:              nextHop ← neighbor;
8:          else if (nextHop.id > value ∧ neighbor.id < value) then
9:              nextHop ← neighbor;
10:         end if
11:     end for
12:     return nextHop;
13: end function
```

Algorithm 10 Range Query

```
1:  procedure Lookup ⟨requester, lookupRequest⟩
2:      if requester ≠ self then
3:          registerNeighborSubscription(requester, lookupRequest)
4:      end if
5:      RVNodes ← NULL
6:      if overlapping(lookupRequest.range, [self, successor]) ≠ ∅ then
7:          RVNodes.add(successor)
8:      end if
9:      for all neighbor in self.neighbors do
10:         if lookupRequest.range.includes(neighbor) then
11:             RVNodes.add(neighbor)
12:         end if
13:     end for
14:     if RVNodes ≠ NULL then                                          ▷ Shower
15:         for all RV in RVNodes do
16:             send lookup(self, lookupRequest) to RV
17:         end for
18:     else                          ▷ Proceed with a greedy lookup for the beginning of the rang
19:         nextHop ← findNextHop(lookupRequest.range.from)
20:         send lookup(self, lookupRequest) to nextHop
21:     end if
22: end procedure
```

Algorithm 11 Install Subscription

```
1:  procedure installSubscription ⟨⟩
                                  ▷ perform a range query for the primary attribute
2:      range ← self.S.getC(primaryAttribute)
3:      lookupRequest ← (primaryAttribute, range, self.S)
4:      lookup(self, lookupRequest)
5:  end procedure
```

nodes is constructed. We refer to this path as the *installation path* (Path $Q \rightarrow R \rightarrow S \rightarrow T$ in Figure 4.2).

Note that in this example there is only one attribute in the system. When there are more attributes, every node only subscribes for the range(s) over its

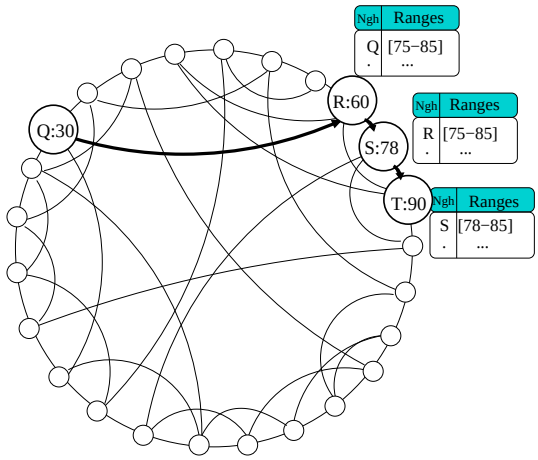


Figure 4.2: Q sends a request for range [75, 85] to its neighbor R. R registers the request in its CRT, and forwards it to neighbor S. S registers the request, and also recognizes that it is responsible for a part of the range ([75, 78]). But the rest of the range still needs to be taken care of. So S forwards the remaining part of request ([78,85]) to T.

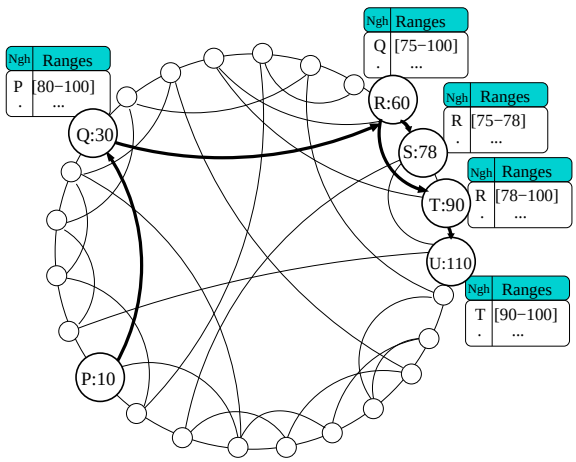


Figure 4.3: Q receives a request for range [80, 100] from node P. It registers the request in its CRT and forwards it to R. R previously had an entry for Q in its CRT. It aggregates the two requests and updates the corresponding entry to range [75, 100]. Then, it forwards the request to its neighbors and the CRT of those nodes are also updated, accordingly.

primary attribute (Algorithm 11). The reason behind this is that the node have more neighbors with overlapping ranges over this attribute, thus, it can benefit more from those neighbors, if it registers its request along a path for this attribute.

However, when sending the lookup request on the primary attribute, a node attaches its whole subscription (on all the attributes) to the request and this subscription is registered in every node along the installation path.

To register the requests, every node maintains a table, called a *Content Routing Table* or *CRT* for short. The CRTs are populated when the queries are forwarded in the overlay. In our example, node R adds an entry to its CRT, for node Q requesting range [75-85], while node S registers range [75, 85] for node R. Finally node T registers a request from node S for the range [78-85]. In this example we only have one attribute, and therefore CRTs, only include the requested range for that single attribute. If there are more attributes, each entry associate the requested range(s) with an attribute. More precisely, an entry of a CRT is a tuple in form of:

$$< Ngh, Attr, Ranges, Subscriptions >$$

where Ngh indicates the neighbor, $Attr$ is the requested attribute, $Ranges$ are the interested ranges over the requested attribute, and $Subscriptions$ are the subscription requests, containing all the attributes, that are received through the neighbor. For example, a sample CRT in a system with two attributes may look like Figure 4.4. This CRT shows that node Q has requested any event that contains a value either in the range [75-100] for attribute A_1, or in the ranges [80-90] or [100-120] for attribute A_2.

Ngh	Attr	Ranges	Subscriptions
Q	A1	[75−100]	**C1:[75−100]** & C2:[10−15]
Q	A2	[80−90]	C1:[200−250] & **C2:[80−90]**
		[100−120]	C1:[50−60] & **C2:[100−120]**
...	...	...	...

Figure 4.4: A sample CRT when the subscription scheme contains two attributes

The subscription installation process is further equipped with an aggregation technique. That is, a node aggregates all the requests it receives from the same neighbor on the same attribute. For example, in Figure 4.3 node P appears in the system and subscribes for the range [80-100]. The closest neighbor of P to the requested range is node Q. So P sends a request to Q. As a result, Q installs this request in its CRT, and forwards it further to node R. Since node R previously had an entry for R in its CRT (for the range [75-85]), it aggregates the two requests and modifies the entry for Q to range [75-100]. Now R knows two nodes, S and T, in the requested range. So it showers the request to both of them, by sending a request for range [75-78] to S and the remaining part of the range to T. When this new request is further forwarded in the overlay, nodes S and T similarly update their CRTs with the range [75, 78] and the aggregated range [78-100], respectively.

Then, node T forwards the request further to node U, which is also a rendezvous node for the requested range.

As mentioned previously, due to the clustering technique used for selecting friend links, nodes with similar subscriptions on the same primary attribute are grouped together. When these nodes install their subscriptions, they initiate a routing towards the same or close-by rendezvous nodes. Therefore, the installation tree is mostly shared between these nodes and the requests can be effectively aggregated. This results in smaller size CRTs, as well as less traffic overhead in the dissemination overlay. Smaller CRTs not only reduce the maintenance cost of the trees, but also simplify the matching process upon receipt of an event.

4.4 Event Dissemination

Any node in Vinifera can publish events. As mentioned previously, an event is a piece of data that has exact values for each attribute. Therefore, in a system with n attributes, an event is associated with n rendezvous nodes, one for each attribute. When a node publishes an event $e\{v_1, v_2, .., v_n\}$, it sends one copy of the event to each of the n relevant rendezvous nodes, i.e., $RV(v_1)$, $RV(v_2)$, .., $RV(v_n)$, which are responsible for the values assigned to each of the attributes. This is done by initiating a simple rendezvous routing for each attribute. Then, n *delivery trees* for the event are constructed on the fly, by following the links on the reverse installation paths from the rendezvous nodes to the subscriber nodes, using the node CRTs.

Figure 4.5 shows an example with two attributes. Two copies of the same event are delivered to the two nodes with the closest, but not higher, identifiers to the the value of each of the attributes. Nodes Q, X, Y, and Z have a matching subscription. Note that each subscriber node is registered in only one of the delivery trees that corresponds to its primary attribute. So, it does not receive redundant messages from multiple trees.

The delivery trees are constructed as follows. Each rendezvous node, matches the event against the subscriptions that are registered in its CRT, and sends the event only to the neighbors with matching subscriptions. Likewise, every node on the path performs such a matching process and forwards the event further, until it reaches the interested subscribers. By this approach, the matching process will be carried out as the event goes down to the subscribers, while every node maintains partial information about the other nodes. In essence, we distribute the load of matching events against subscriptions between the nodes that are on the installation path. At every step, those branches, which are not interested in the event, are pruned and the event is forwarded only down the paths that hold some interested node(s).

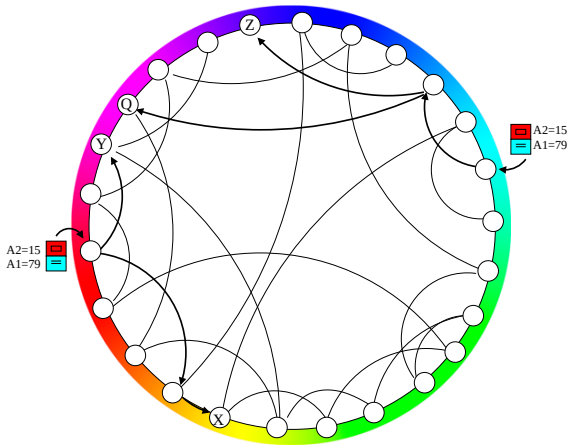


Figure 4.5: A new event is published in a system with two attributes: A_1 and A_2. Nodes Q, X, Y, and Z have a matching subscription, which they have already installed in the CRT of their neighbors. The primary attribute for nodes X and Y is A_2, and for nodes Q and Z is A_1. The event is delivered to two rendezvous nodes, one for each of the attributes. Nodes X and Y receive the event from a delivery tree rooted at the rendezvous node for the value of A_2, while Q and Z receive it from another delivery tree that is rooted at a rendezvous node for the value of attribute A_1.

4.5 Load Balancing

Due to the prevalent skewed subscription patterns in the real world, the use of an Order Preserving Hash Function (OPHF) inevitably results in a non-uniform identifier distribution and thus, an unbalanced load on the nodes. More precisely, some regions in the identifier space might be very popular with huge number of corresponding events, while some other regions might not be popular at all. So, the nodes who happen to fall into the popular regions may have to deal with huge number of requests. For example, if an attribute in the subscription scheme is temperature in a city, then the published events are most likely in the range $[-10, +30]$, probably a few around this range, and almost no event in less than -30 or over $+50$. Hence, node that have an identifier between $OPHF(-10)$ and $OPHF(+30)$ are likely to be highly loaded, while the rest of the nodes are under loaded.

In order to alleviate this problem, we utilize an idea, inspired by [68], for adapting the node identifier to the existing load in the network. The idea is that Vinifera nodes, can change their identifiers along the ring, while their connections remains intact. In other word, nodes only change their identifier to an identifier between themselves and their successor. The new identifier is assigned to the node to halve the load on the successor. Since nodes do not change their neighbors upon change of the identifier, they can easily inform their neighbors of the new identifier, when they send the next gossip message to the neighbors.

Algorithm 12 Load Balancing

```
1:  procedure BalanceLoad ⟨⟩
2:      if (self.load = 0 ∧ successor.load > β ∨
3:        (successor.load > self.load ∗ β) then
4:          if successor.id = 0 then
5:              sum ← 2^15 + self.id                ▷ 2^15 is the size of the identifier space
6:          else
7:              if successor.id < self.id then
8:                  sum ← successor.id
9:              else
10:                 sum ← successor.id + self.id
11:             end if
12:         end if
13:         self.id ← sum / 2                       ▷ update node identifier
14:         newLoad ← (successor.load + self.load ) / 2
15:         self.load ← newLoad
16:         successor.load ← newLoad
17:     end if
18: end procedure
```

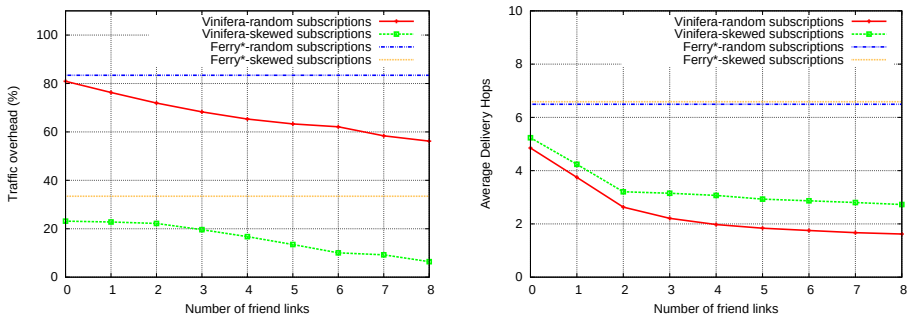
For our system we define a measure of load as follows. Every node counts the number of events that it receives as a rendezvous node, without having any interest in them. Whenever the node sends its gossip message to its predecessor, it piggybacks its current load on the message. In order to prevent perturbation of the node identifiers, we define a threshold β for load imbalance between a node and its successor. When the difference of load between the two nodes exceeds the threshold β, the proceeding node changes its identifier to a value closer to its successor. This value is selected, such that half of the responsibility range of the successor node is delegated to the node (See Algorithm 12). Then the two nodes update their load to the average of their current loads. We show in the evaluation section that by employing this mechanism, Vinifera nodes can adapt to even very skewed user subscriptions.

4.6 Experimental Results

4.6.1 Experimental Environment and Settings

We implemented Vinifera in Peersim [65], which is a discrete event simulator for peer-to-peer applications. Through extensive simulations we evaluated the performance of Vinifera, while comparing it against a system equivalent to Ferry [12], a state-of-the-art solution that is able to scale to a large number of nodes and attributes, due to the bounded node degree requirement. To make comparisons fair, we implemented this Ferry-like system with the same gossiping protocol that we used for Vinifera. To distinguish between the implemented system in [12] and our implementation of it, hereafter, we refer to the latter as Ferry*.

We synthetically generated two different user subscriptions models: random subscriptions, and skewed subscriptions. In the random subscription model, every



(a) Traffic Overhead (b) Delivery Latency

Figure 4.6: Measurements with varying number of friend links

user selects a range in the attribute space, uniformly at random. However, in the skewed model, the range selection follows a Zipf distribution. That means some ranges in the attribute space are more popular than others.

Moreover, any node can be a publisher. The publishers are selected randomly in each round. Any publisher, only generates events, in which it is interested itself. That means the event distribution follows a distribution similar to that of the subscriptions.

In our experiments we focus on four different aspects of the systems: scalability, load balancing, working under heavy workload, and fault tolerance. The metrics that we use are the followings:

- Hit Ratio: The fraction of events that are received by the matching subscribers.
- Traffic Overhead: The fraction of message traffic generated in the overlay that is not interesting for the nodes.
- Delivery Latency: The average number of hops that takes for the events to reach the matching subscribers.

The size of the identifier space is set to 2^{15}, the parameters of the Zipf distribution are $size = 32$ and $skew = e^{-1}$. Unless otherwise mentioned, the network size $N = 1000$ and the overlay membership is static, i.e., there is no joins or failures.

4.6.2 Topology Configuration

In this experiment, we investigate how Vinifera performs under different topology configurations. Two main parameters that affect the topology in Vinifera are K_{sw}, and K_f, which define the number of small-world links and friend links, respectively. We run this experiment with 1000 nodes and set the total number of links per node

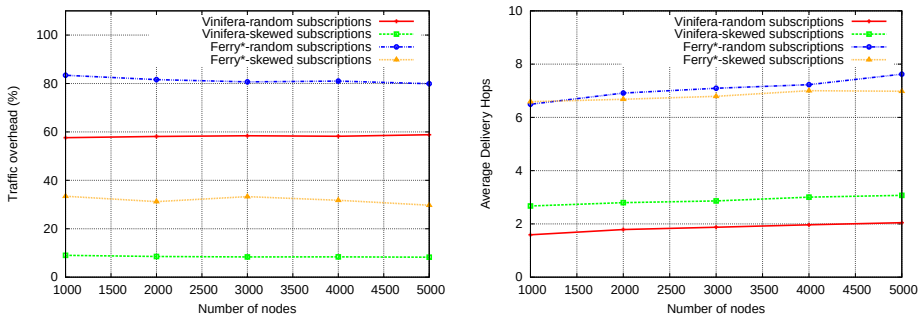
to 10 (i.e., $\sim log(N)$). From these 10 links, two are dedicated to ring links and the rest are used for small-world and friend links, i.e., $K_{sw} + K_f = 8$.

Figure 4.6a shows how the generated traffic overhead changes when K_f increases. In this figure we also compare Vinifera to a pure small-world solution, such as Ferry*, where no friend link exists. When the subscriptions are random, i.e., the worst case scenario, the generated traffic overhead in Ferry* is over 80%. Although the similarity of subscriptions is the least in the random subscriptions scenario, Vinifera can still benefit from adding more friend links (i.e. less small-world links) by exploiting even the slightest correlations between user interests. A similar trend is observed with skewed subscriptions. The reason for a reduction in the level of traffic overhead in both systems, compared to the random scenario, is that when some region in the attribute space is more popular, then more events are published in that region. As a result, most of the published events are interesting for most of the users, and there are very few nodes that consider the popular events as overhead.

We also measured the average delivery hops with different number of small-world links in Figure 4.6b. As it is shown, even the delivery latency improves when there are more friend links. This is due to the better clusterization of the nodes that results in the construction of shorter trees in Vinifera, as opposed to the single long tree in Ferry*. We conclude that by employing an appropriate clustering technique together with constructing many small trees, Vinifera not only generates less traffic, but also provides faster delivery. Therefore, for the rest of our experiments, we prioritize the friend links over small-world links and set K_{sw} to 0 and K_f to $log(N) - 2$ (two links are used for the ring), where N is the number of nodes in the network.

4.6.3 Scalability

To measure the scalability of Vinifera, we performed experiments with different number of nodes, as well as, different number of attributes. Due to the static overlay membership in this experiment, both Vinifera and Ferry* had a full hit ratio in all network sizes and subscription schemes. Figure 4.7a shows the traffic overhead of both systems. The performance of both systems is almost the same for different network sizes. However, the traffic overhead in Ferry* is more than 80% for random subscriptions, whereas it is reduced to 60% in Vinifera. Note that random subscriptions bring up the worst case scenario, for nodes can not effectively benefit from our clustering technique, due to lack of any correlation between user subscriptions. However, it is shown that a significant subscription correlation exist in real-world application [53] [54]. When the user subscriptions are skewed, the traffic overhead in Ferry* drops to nearly half. This is mainly due to the skewed distribution of events, which generates more matching events in the system. In the same scenario, Vinifera reduces the traffic to almost one sixth of that of the random subscriptions. This shows that our data dissemination overlay is remarkably benefiting from the utilized clustering technique. Moreover, the traffic



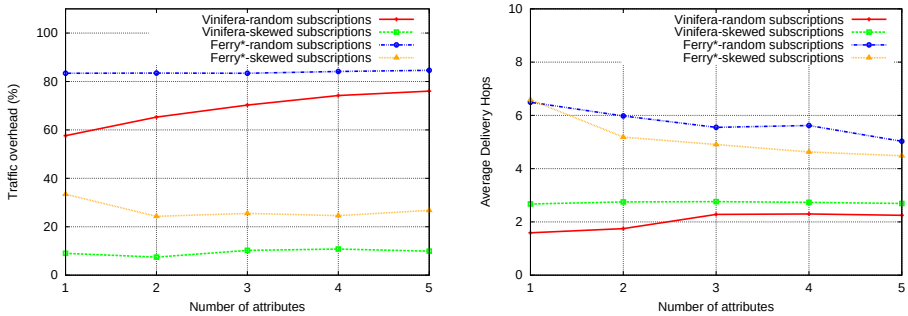
(a) Traffic Overhead (b) Delivery Latency

Figure 4.7: Scalability with the number of nodes

overhead in Vinifera is reduced to almost one third of that of Ferry* with skewed subscription pattern.

Figure 4.7b shows the delivery latency of both systems in terms of hop counts. Here again, the number of hops in Vinifera is nearly one third of that of Ferry*. However, the delivery latency is slightly higher with skewed subscriptions, because the overlay topology is clustered. Whereas, in the random subscription model the overlay better resembles a random network.

Next, we observe the behavior of both systems when the subscription model includes more attributes. We increase the number of attributes from 1 to 5 and measure the traffic overhead and delivery latency of the two systems. Figure 4.8a shows that when the subscription model is random, the traffic overhead of Ferry* remains at around 80%. This overhead, however, increases in Vinifera, because when there are more attributes, nodes select different primary attributes, and in a random subscription scheme there exists very little similarities to be exploited by this primary attribute. In practice, the primary attribute in such a scenario is a random attribute for each node. As a result the installation paths, and thus, the delivery trees are scattered in the overlay and nodes can not effectively cooperate in data dissemination. However, this overhead is still less than that of Ferry*. With the skewed subscription model, both systems behave significantly better. The overall improvement is again due to the skewed event publication in the system. However, the improvement in Ferry* with more attributes is because instead of having one single tree, more delivery trees are constructed, one for each attribute. Each node joins one of these trees, as a subscriber, and does not receive the events that are forwarded on other trees, unless it is a relay node in those trees. Moreover, nodes with similar subscriptions are grouped together, and the delivery tree is shared between these nodes. Thus, the overall number of uninterested node on the delivery trees is reduced and that is why the traffic overhead in Vinifera drops to nearly one third of that of Ferry*. The delivery latency of the two systems is shown



(a) Traffic Overhead (b) Delivery Latency

Figure 4.8: Scalability with the number of attributes

in Figure 4.8b. Ferry* deliver the events slightly faster when there are more attributes, due to a policy that is employed to select as the primary attribute, the one that has the closest rendezvous node to the subscriber node in the identifier space. However, the delivery latency in Vinifera is still by far better than Ferry*, even with 5 attributes in the subscription model, thanks to the utilized clustering technique. As soon as an event reaches a cluster of nodes with matching subscriptions, it is propagated inside that cluster very quickly.

We conclude that although both systems can accommodate any number of attributes in the subscription scheme, Vinifera exhibits a significantly better performance than Ferry*, specially in the presence of skewed subscriptions.

4.6.4 Load Balancing

In this section, we explore how the load is distributed among the node in Vinifera versus Ferry*. For this purpose we plot the cumulative distribution of load on the nodes, for 1, 2, and 3 attributes, and report the results in Figure 4.9.

Although with more attributes, the load distribution is slightly degraded in Vinifera, it is still significantly better than Ferry* and the load on any Vinifera node never exceeds 30%. More precisely, over 95% of the nodes has a load less than 20%, even with 3 attributes, whereas, 10% of Ferry* nodes suffer from over 60% load in the system, while nearly 40% of the nodes have zero load. Figure 4.9 shows that the load in Ferry* is extremely unbalanced. This is because nodes up on the delivery tree are highly stressed, while leaf nodes are just receiving the service for free. There are nodes with even nearly 100% load (the rendezvous nodes), which can significantly harm the performance of the system as soon as they stop functioning correctly, due to congestion or failure. This problem prevents Ferry* to work under real-life scenarios where node and network failures are inevitable, while Vinifera can still function without having any imminent bottleneck.

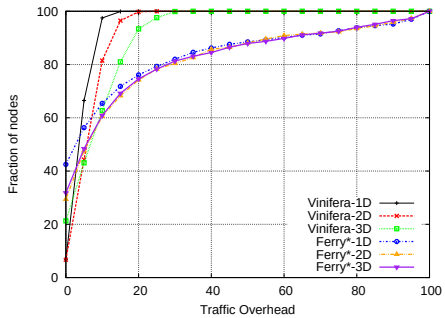


Figure 4.9: Load Distribution in Vinifera vs. Ferry*

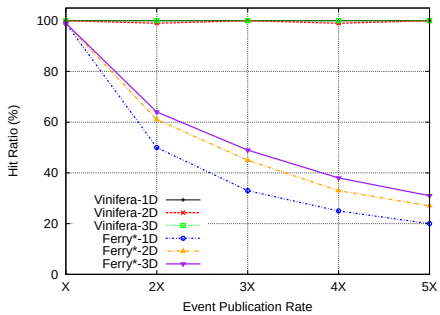


Figure 4.10: Hit Ratio for different publication rates

4.6.5 Workload

In this section we examine if the systems can function under different workloads, i.e., under different event publication rates. To model the congestion, we assume that every node can handle a bounded number of messages, X, in every round, and if it receives more events it will simply drop them. Then, we increase the number of events that are published in the system to up to five times X.

Figure 4.10 shows that the hit ratio in Ferry* significantly drops as soon as the publication rate passes the X threshold. Whereas, Vinifera survives even under high event publication rates. This is due to the fact that Ferry* relies on very few nodes to propagate the event in the system (only the intermediary nodes in the delivery tree). Therefore, under a high publication rate, those nodes become highly overloaded and start dropping the messages. When a node in the tree drops a message, all its descendant nodes fail to receive the message. On the other hand, in Vinifera load is almost evenly distributed among the nodes. Thus, the nodes do not have to drop the messages due to the excessive load.

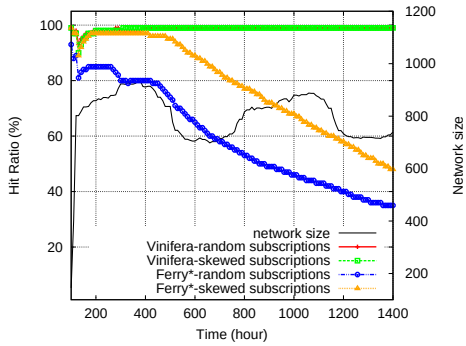
4.6.6 Fault Tolerance

To evaluate the performance of Vinifera in the presence of failures, we used real-world churn traces [8], that were obtained by monitoring a set of 4000 nodes participating in the Skype superpeer network for one month beginning September 12, 2005.

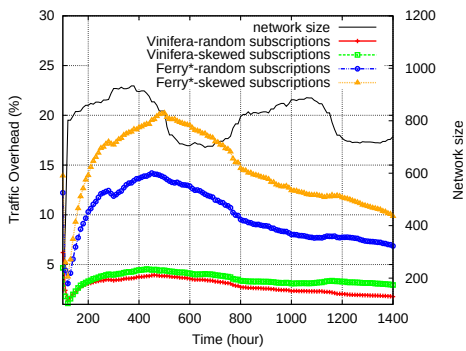
In Figure 4.11 the x-axis shows the time, while the y-axis on the right shows the network size. The black solid line in the three graphs shows how the network size changes over time. Figure 4.11a shows the hit ratio of the two systems with random and skewed subscriptions. Although the hit ratio of Vinifera slightly decreases in the flash crowds, i.e., around time 100 h., when a large number of nodes join the system concurrently, the system recovers quickly and the hit ratio goes back to and remain at 100%, even in the presence of further joins and failures. In contrast, the hit ratio in Ferry* is highly affected by churn, due to the fragile structure of a single delivery tree. When this tree is broken, Ferry* can not repair it quickly enough to catch up with further event deliveries. If the network remain stable for a while, that is when no more node joins or fails, Ferry* is able to repair the dissemination tree. However, this real-world trace suggests that such stable periods are hardly happening.

Figure 4.11b shows the traffic overhead in both systems. The traffic overhead in Vinifera is one forth compared to Ferry* for both random and skewed subscriptions. Note that the decrease in the traffic overhead of Ferry* is mainly due to the less than full hit ratio in this system. In other words, since there are parts of the overlay that fail to receive the events, the traffic overhead in those parts, and therefore, the average traffic overhead in the network is reduced.

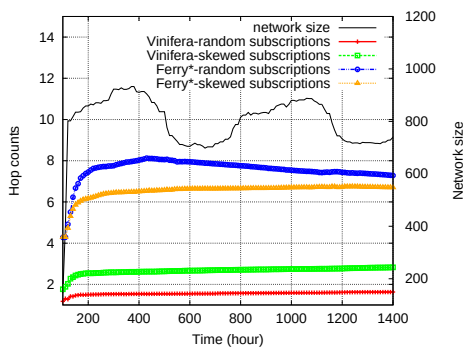
We also measured the delivery latency in both systems and observed, in Figure 4.11c, that Vinifera is four times faster than Ferry*, in the presence of churn. Here again, we should take into account that not all the Ferry* nodes are receiving the events, and the measured values for Ferry* only includes the nodes that received the events.



(a) Hit ratio



(b) Traffic overhead



(c) Delivery latency

Figure 4.11: Performance result of Vinifera Vs. Ferry* in the presence of churn

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this work, we introduced two systems, namely Vitis and Vinifera, for topic-based and content-based publish/subscribe models, respectively.

We showed that Vitis fills in the gap in the range of topic-based solutions, by simultaneously achieving both bounded node degree and low traffic overhead. Vitis constructs a novel hybrid publish/subscribe overlay that exploits two ostensibly opposite mechanisms: unstructured clustering of similar peers and structured rendezvous routing. We employed a gossiping technique to embed a navigable small-world network, which efficiently establishes connectivity among clusters of nodes that exhibit similar subscriptions. We also gave a theoretical bound on the worst case delay. To reduce the traffic overhead, we introduced a novel technique for selecting representative nodes that connect together all the subscribers of a topic across the clusters.

We evaluated Vitis, in simulations, by comparing its performance against two base-line solutions, which represent two main groups of the related work: a structured overlay that uses rendezvous routing (RVR), and a solution that takes advantage of subscription similarities to constructs an overlay per topic (OPT). We used synthetic data as well as real-world traces from Twitter to model users subscriptions. We also used traces from Skype to show that Vitis is robust in the presence of churn. We showed that although exploiting subscription correlations results in great advantages, solutions such as OPT, which solely rely on such correlations, can not scale when the number of node subscriptions increases. Consequently, in real world scenarios, such solutions cannot guarantee that the subscribers receive their intended data, unless the node degrees are unbounded. In contrast, Vitis and RVR always reach a perfect hit ratio. This, however comes at the cost of some traffic overhead. We showed that, compared to RVR, Vitis reduces the traffic overhead

to less than 75% with synthetic data and 40% for real-world traces, while it speeds up the event dissemination in the overlay. Moreover, Vitis adapts to biased rates of events that are published on different topics, and builds more efficient groups for hot topics, thus, improving the overall performance of the event dissemination.

We also introduced Vinifera, a peer-to-peer content-based publish/subscribe system that enables users to define a filter over the content of the information they are willing to receive. We employed a gossip-based technique to construct a topology that not only resembles a small-world network, but also puts the nodes with similar interests close to one another by using a clustering technique for capturing the existing correlation between user subscriptions. On top of this hybrid overlay, we utilized a rendezvous routing mechanism to propagate node subscriptions in the overlay. Together with an order preserving hashing technique and an efficient showering algorithm we enabled range queries, and at the same time, we employed a load balancing technique to deal with the potential non-uniform user subscriptions. Furthermore, we used an aggregation technique to simplify the matching of events against subscriptions, and reduced the maintenance cost of the event dissemination paths. By this approach we constructed efficient data delivery paths in the overlay. When events are published, multiple copies are disseminated in the overlay. Events are forwarded along the reverse paths, on which the subscriptions are propagated, while the matching is partially done by the nodes along the paths.

We showed, in our experiments, that Vinifera scales with both the number of nodes in the network and the the number of attributes in the subscription scheme. It generates only one third of the traffic overhead that is observed in the existing state-of-the-art systems, while the events are delivered up to four times faster. Furthermore, we showed that Vinifera balances the load of subscription maintenance, matching, and event forwarding among the participating nodes. This is of critical importance, because it enables Vinifera to function under very high workload, i.e., when events are published at a high rate. We modeled congestion in the network and showed that event delivery is still guaranteed, even under high event publication rates. We also used Skype traces to model churn in the network, and showed that Vinifera is very robust in the presence of failures.

To summarize, the work on pub/sub carried out within this thesis revealed the huge potential of combining two conceptually different paradigms: unstructured gossiping and structured Small-World navigability. Such merger of both unstructured and structured concepts into a single hybrid system provided us with an opportunity to build novel pub/sub systems that could extract the best traits of the existing pub/sub techniques, namely: rendezvous routing and the gossip-based clusterization. Our hybrid pub/sub systems exhibited superior performance against the state-of-the-art techniques, effectively without the need to trade-off or degrade any important properties of the system. The overlay topologies autonomously adapt to the user subscriptions and are highly resilient to the dynamism in the network.

The generated traffic overhead and the delivery latency are simultaneously kept low in both works, while only a bounded node degree is required and no global knowledge at any point is assumed.

5.2 Future Work

We will continue our work on the aforementioned topics and plan to cover the following research directions:

- **Durability.** While Vitis and Vinifera can guarantee w.h.p. the event delivery to the on-line users, the next step to further enhance them, is to augment them with proper caching and replication mechanisms, such that the durability of messages is also guaranteed. Meaning that, if a user goes off-line and comes back again, it receives all the messages that were published during its absence period.

- **Locality Awareness.** The utility functions that we introduced for Vitis and Vinifera can be further improved to account for proximity in the network. This could be of interest for many applications that work across ISPs and would like to confine the traffic to inside the ISPs, as much as possible.

- **Load Balancing.** We believe that our load-balancing technique, introduced in Vinifera, can be used in other overlay networks for evenly distributing the load of the systems across nodes. Hence, more investigation on the trade-offs for different threshold values of β, is of great interest. Moreover, this technique can be further improved to speed up the load balancing process.

- **Real-World Evaluations.** One of our future plans is to evaluate the performance of Vitis and Vinifera over a real network environment. For this purpose, we also would like to use subscription and event traces from real applications. Most of the related work, specially for content-based systems, only use synthetically generated workload. We are now trying to reach out to some real-world workloads for our further evaluations.

- **Fuzzy Publish/Subscribe.** The notion of *implicit subscriptions* is recently gaining interest and is discussed in some works such as [69] and [70]. One research direction is, therefore, to address the challenges that arise in systems with such subscriptions. The subscription model can be extended to *fuzzy subscriptions*, in which users do not have to express their interests precisely in accordance to a predefined scheme, and the subscriptions are modeled by a probability density function over the event space.

Bibliography

[1] "IBM Smarter Planet," *URL: http://www. ibm. com/ smarterplanet/ us/ en*, June 2011.

[2] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, 2003.

[3] F. Rahimian, S. Girdzijauskas, A. Payberah, and S. Haridi, "Vitis: A gossip-based hybrid overlay for internet-scale publish/subscribe enabling rendezvous routing in unstructured overlay networks," in *2011 IEEE International Parallel & Distributed Processing Symposium*, 2011.

[4] R. Baldoni and A. Virgillito, "Distributed event routing in publish/subscribe communication systems: a survey," *DIS, Universita di Roma La Sapienza, Tech. Rep*, 2005.

[5] J. Kleinberg, "The small-world phenomenon: an algorithm perspective," pp. 163–170, 2000.

[6] S. Girdzijauskas, A. Datta, and K. Aberer, "On small world graphs in non-uniformly distributed key spaces," in *Data Engineering Workshops, 2005. 21st International Conference on*, pp. 1187–1187, IEEE, 2005.

[7] W. Galuba, K. Aberer, D. Chakraborty, Z. Despotovic, and W. Kellerer, "Outtweeting the Twitterers-Predicting Information Cascades in Microblogs," in *3rd Workshop on Online Social Networks (WOSNâ2010)*, 2010.

[8] S. Guha, N. Daswani, and R. Jain, "An experimental study of the skype peer-to-peer voip system," in *Proc. of IPTPS*, vol. 6, Citeseer, 2006.

[9] E. Fox, Q. Chen, A. Daoud, and L. Heath, "Order-preserving minimal perfect hash functions and information retrieval," *ACM Transactions on Information Systems (TOIS)*, vol. 9, no. 3, pp. 281–308, 1991.

[10] Z. Czech George and B. Majewski, "An optimal algorithm for generating minimal perfect hash functions," *Information Processing Letters*, 1992.

[11] S. Girdzijauskas, "Designing peer-to-peer overlays : a small-world perspective," *EPFL thesis no. 4327, advisor: Karl Aberer*, 2009.

[12] "Ferry: An architecture for content-based publish/subscribe services on p2p networks," in *Proceedings of the 2005 International Conference on Parallel Processing*, IEEE Computer Society, 2005.

[13] Y. Diao, S. Rizvi, and M. Franklin, "Towards an internet-scale xml dissemination service," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, VLDB, 2004.

[14] E. Grummt, "Fine-grained parallel xml filtering for content-based publish/-subscribe systems," in *Proceedings of the 5th ACM international conference on Distributed event-based system*, ACM, 2011.

[15] I. Miliaraki, Z. Kaoudi, and M. Koubarakis, "Xml data dissemination using automata on top of structured overlay networks," in *Proceeding of the 17th international conference on World Wide Web*, ACM, 2008.

[16] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[17] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, vol. 11, pp. 329–350, Citeseer, 2001.

[18] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," *Peer-to-Peer Systems*, pp. 53–65, 2002.

[19] G. Manku, M. Bawa, and P. Raghavan, "Symphony: Distributed hashing in a small world," in *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems-Volume 4*, USENIX Association, 2003.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 161–172, ACM, 2001.

[21] B. Leong and J. Li, "Achieving one-hop dht lookup and strong stabilization by passing tokens," in *Networks, 2004.(ICON 2004). Proceedings. 12th IEEE International Conference on*, vol. 1, pp. 344–350, IEEE, 2004.

[22] S. Girdzijauskas, A. Datta, and K. Aberer, "Oscar: Small-world overlay for realistic key distributions," in *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, pp. 247–258, Springer-Verlag, 2005.

[23] S. Girdzijauskas, A. Datta, and K. Aberer, "Oscar: A data-oriented overlay for heterogeneous environments," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 1365–1367, IEEE, 2007.

[24] Š. Girdzijauskas, A. Datta, and K. Aberer, "Structured overlay for heterogeneous environments: Design and evaluation of oscar," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 5, no. 1, p. 2, 2010.

[25] *URL: http://www. gnutella. com*, 2006.

[26] *URL: http://www. kazaa. com*, March 2009.

[27] J. Kleinberg, "Navigation in a small world," *Nature*, vol. 406, no. 6798, pp. 845–845, 2000.

[28] A. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," in *ACM SIGCOMM Computer Communication Review*, vol. 34, pp. 353–366, ACM, 2004.

[29] P. Eugster, R. Guerraoui, S. Handurukande, P. Kouznetsov, and A. Kermarrec, "Lightweight probabilistic broadcast," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 4, pp. 341–374, 2003.

[30] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 3, pp. 219–252, 2005.

[31] M. Jelasity and O. Babaoglu, "T-Man: Gossip-based overlay topology management," *Lecture Notes in Computer Science*, vol. 3910, p. 1, 2006.

[32] A. Payberah, J. Dowling, F. Rahimian, and S. Haridi, "gradientv: Market-based p2p live media streaming on the gradient overlay," in *Distributed Applications and Interoperable Systems*, pp. 212–225, Springer, 2010.

[33] A. Payberah, J. Dowling, F. Rahimian, and S. Haridi, "Sepidar: Incentivized market-based p2p live-streaming on the gradient overlay network," in *International Symposium on Multimedia, vol. 0*, pp. 1–8, 2010.

[34] A. Payberah, J. Dowling, and S. Haridi, "Glive: The gradient overlay as a market maker for mesh-based p2p live streaming," in *the 10th IEEE International Symposium on Parallel and Distributed Computing (ISPDC)*, 2011.

[35] M. Jelasity, S. Voulgaris, R. Guerraoui, A. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 3, p. 8, 2007.

[36] M. Jelasity, A. Kermarrec, and M. van Steen, "The peer sampling service: Experimental evaluation of unstructured gossip-based implementations," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pp. 79–98, Springer-Verlag New York, Inc. New York, NY, USA, 2004.

[37] A. Payberah, J. Dowling, and S. Haridi, "Gozar: Nat-friendly peer sampling with one-hop distributed nat traversal," in *Distributed Applications and Interoperable Systems*, pp. 1–14, Springer, 2011.

[38] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer, "Brahms: byzantine resilient random membership sampling," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pp. 145–154, ACM, 2008.

[39] A. Ganesh, A. Kermarrec, and L. Massoulié, "Scamp: Peer-to-peer lightweight membership service for large-scale group communication," *Networked Group Communication*, pp. 44–55, 2001.

[40] M. Jelasity, W. Kowalczyk, and M. Van Steen, "Newscast computing," *submitted for publication*, vol. 266, pp. 268–269.

[41] S. Voulgaris, D. Gavidia, and M. Van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[42] A. Carzaniga, D. Rosenblum, and A. Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, p. 227, ACM, 2000.

[43] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward, "Gryphon: An information flow based approach to message brokering," in *International Symposium on Software Reliability Engineering*, Citeseer, 1998.

[44] P. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," 2002.

[45] V. Ramasubramanian, R. Peterson, and E. Sirer, "Corona: A high performance publish-subscribe system for the world wide web," *Proceedings of Networked System Design and Implementation (NSDI)*, 2006.

[46] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[47] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," in *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, p. 20, ACM, 2001.

[48] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "TERA: topic-based event routing for peer-to-peer architectures," in *Proceedings of the international conference on Distributed event-based systems*, ACM, 2007.

[49] J. Patel, É. Rivière, I. Gupta, and A. Kermarrec, "Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems," *Computer Networks*, vol. 53, no. 13, pp. 2304–2320, 2009.

[50] M. Matos, A. Nunes, R. Oliveira, and J. Pereira, "StAN: Exploiting Shared Interests without Disclosing Them in Gossip-based Publish/Subscribe," *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2010.

[51] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication," p. 25, 2007.

[52] B. Wong and S. Guha, "Quasar: A Probabilistic Publish-Subscribe System for Social Networks," in *Proceedings of The 7th International Workshop on Peer-to-Peer Systems (IPTPS 2008), Tampa Bay, FL (February 2008).*

[53] H. Liu, V. Ramasubramanian, and E. Sirer, "Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews," in *Proc. of ACM Internet Measurement Conference*, 2005.

[54] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky, "Hierarchical clustering of message flows in a multicast data dissemination system," in *IASTED PDCS*, pp. 320–326, 2005.

[55] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," *Computer*, vol. 74, pp. 11–20, 2001.

[56] S. Girdzijauskas, G. Chockler, Y. Vigfusson, Y. Tock, and R. Melamed, "Magnet: practical subscription clustering for Internet-scale publish/subscribe," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, ACM, 2010.

[57] S. Girdzijauskas, G. Chockler, R. Melamed, and Y. Tock, "Gravity: An interest-aware publish/subscribe system based on structured overlays," in *LADIS2008: Proceedings of the 2nd Large-Scale Distributed Systems and Middleware Workshop*, 2008.

[58] J. Alveirinho, J. Paiva, J. Leitao, and L. Rodrigues, "Flexible and Efficient Resource Location in Large-Scale Systems," *The 4th ACM SIGOPS/SIGACT Workshop on Large Scale Distributed Systems and Middleware*, 2010.

[59] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "TERA: topic-based event routing for peer-to-peer architectures," p. 13, 2007.

[60] A. Gupta, O. Sahin, D. Agrawal, and A. Abbadi, "Meghdoot: content-based publish/subscribe over P2P networks," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Springer-Verlag New York, Inc., 2004.

[61] S. Voulgaris, E. Riviere, A. Kermarrec, and M. Van Steen, "Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks," in *IPTPSâ2006: the fifth International Workshop on Peer-to-Peer Systems*, Citeseer, 2006.

[62] R. Zhang and Y. C. Hu, "Hyper: A hybrid approach to efficient content-based publish/subscribe," in *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05, (Washington, DC, USA), IEEE Computer Society, 2005.

[63] X. Yang, Y. Zhu, and Y. Hu, "Scalable content-based publish/subscribe services over structured peer-to-peer networks," in *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '07, (Washington, DC, USA), pp. 171–178, IEEE Computer Society, 2007.

[64] J. Pujol Ahullo, P. Garcia Lopez, and A. F. Gomez Skarmeta, "Towards a lightweight content-based publish/subscribe services for peer-to-peer systems," *Int. J. Grid Util. Comput.*, 2009.

[65] M. Jelasity, A. Montresor, G. Jesi, and S. Voulgaris, "PeerSim: A peer-to-peer simulator," *URL: http://peersim. sourceforge. net.*

[66] T. Wong, R. Katz, and S. McCanne, "An evaluation of preference clustering in large-scale multicast applications," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, IEEE, 2002.

[67] M. Kurant, A. Markopoulou, and P. Thiran, "On the bias of BFS," *Arxiv preprint arXiv:1004.1729*, 2010.

[68] D. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, ACM, 2004.

[69] S. Buchegger, D. Schiöberg, L. Vu, and A. Datta, "Peerson: P2p social networking: early experiences and insights," in *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pp. 46–52, ACM, 2009.

[70] K. Pripužić, I. Žarko, and K. Aberer, "Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w," in *Proceedings of the second international conference on Distributed event-based systems*, pp. 127–138, ACM, 2008.