UiO : Department of Informatics
University of Oslo

Visualizing and Evaluating the Performance of Overlay-Based Pub/Sub Systems

Nils Peder Korsveien
master thesis autumn 2014

Visualizing and Evaluating the Performance of Overlay-Based Pub/Sub Systems

Nils Peder Korsveien

9th August 2014

Abstract

The Publish/Subscribe (pub/sub) paradigm is receiving an increasing amount of attention both in research and in industry, as it provides a loosely coupled and scalable interaction scheme suitable for large-scale distributed systems. Designers of such system face many difficulties related to finding the correct balance of performance, as many desired properties of pub/sub systems are at stake with each other. Also, many considerations must be taken with regards to the overlays formed by these systems both in terms of maintenance of the overlay as well as structural properties such as clustering and connectivity. With these issues in mind, we propose VIZPUB, a tool for visualizing the performance of overlay-based pub/sub systems. VIZPUB is able to log local information from every node participating in the overlay, and collect this information to a single site. Then, our tool is able to calculate certain system-wide metrics and produce step-by-step visualizations of the overlay in question. To the best of our knowledge, VIZPUB is the first tool of its kind. It is a generic tool intended to be used with any overlay-based system. We describe the system architecture and explore the capabilities of VIZPUB by providing examples of visualizations and what information we may derive from them. Also, we test the potential of using our tool as a convenient framework for evaluating pub/sub system on a set of specific overlay topology metrics. We presented a poster and held a live demonstration of VIZPUB at the ACM International Conference of Distributed Event Based Systems (DEBS), hosted in Mumbai in May 2014. Here, the value of our contribution was confirmed as we were voted best poster/demonstration by the participants attending the conference. Due to interest in our tool, we choose to make our implementation code open source and available in a public repository.[1] Hopefully, VIZPUB may serve as a lasting contribution to the research community.

[1] htttp://github.com/vizpub/vizpub

Contents

1 Introduction **1**
1.1 Chapter Overview 3

2 Background **5**
2.1 The Peer-to-Peer Network Architecture 5
2.2 Overlays 5
2.3 The Publish-Subscribe Communication Paradigm 6
2.3.1 Message Filtering in Pub/Sub 7
2.4 The Gephi Open Graph Viz Platform 8
2.5 The Gephi Toolkit 9
2.6 The GEXF File format 9
2.6.1 Dynamics 10
2.7 Pub/Sub Protocols 11
2.7.1 PolderCast 11
2.7.2 Scribe 13
2.8 The PeerNet Simulator 14
2.9 Description of important metrics 14
2.9.1 Structural overlay properties 14
2.9.2 Dissemination properties 15
2.9.3 Maintenance overhead 16
2.9.4 Using Test-Driven Development 17
2.9.5 Sharing Code with the Community 17

3 Design Challenges in Topic-Based Pub/Sub **19**
3.1 Desired System Properties 19
3.2 Handling Trade-offs 21
3.2.1 Overlay construction 22
3.3 Event dissemination 24
3.4 Chapter Summary 26

4 Visualizing Performance in Overlay-based Pub/Sub Systems **27**
4.1 System Architecture 29
4.1.1 Reporter 30

4.1.2 Collector . . . 32
4.1.3 Visualization Unit . . . 34
4.2 Examples of Visualizations . . . 42
4.2.1 Data Traces Used in Simulations . . . 42
4.2.2 Overlay Evolution During Churn . . . 43
4.2.3 Visualizing Performance Metrics . . . 43
4.2.4 Publication Message Dissemination . . . 45
4.2.5 Comparing Pub/Sub Systems Visually . . . 50
4.3 Implementation Work . . . 58
4.3.1 Using Visualizations to Analyze Datasets . . . 58
4.3.2 Debugging Pub/Sub Systems Visually . . . 58
4.4 Chapter Summary . . . 60

5 Testing and Evaluation **61**
5.1 VIZPUB as a Framework for Evaluating Pub/Sub Systems . . . 61
5.2 Experimental Restrictions . . . 62
5.3 Experimental Setup . . . 63
5.4 Results . . . 63
5.5 Summary . . . 65

6 Conclusion and Future Work **69**
6.1 Conclusion . . . 69
6.2 Future Work . . . 71
6.2.1 Further Evaluation . . . 71
6.2.2 Improving Report File Sizes . . . 71
6.2.3 Implementing Database Support . . . 71
6.2.4 Gephi Performance Issues . . . 72
6.2.5 Reporter Performance Issues . . . 72
6.2.6 Visualizing with Custom Colors and Sizes . . . 72
6.2.7 Collector Scalability . . . 73
6.2.8 Including Associative Arrays in Gephi . . . 73
6.2.9 Implementing Global Attribute Visualization . . . 73
6.2.10 VIZPUB as an Interactive Monitoring System . . . 73
6.2.11 Playback issues in the Timeline component . . . 74

List of Figures

2.1 The basic architecture of a pub/sub system. 6
2.2 The Gephi Tool supports visualization of graphs through coloring and sizing the visual graph representation. It also enables adding labels to nodes and edges. In this screenshot, Gephi is used to detect and visualize communities. 8
2.3 A GEXF description of a minimal static graph 10
2.4 Example of dynamic GEXF file using spells 10
2.5 PolderCast consists of several decoupled components where a dissemination overlay is constructed at the RINGS layer 11
2.6 PolderCast utilizes a hybrid dissemination scheme when publishing messages, relying both on ring links as well as random links 12
2.7 From left to right, the structure of a routing table in Pastry and an example of the Pastry routing scheme, where a message from node *65a1fc* with key *d46a1c* is delivered to the rendezvous node (figures borrowed from [6]) 13

4.1 Visualization of disconnected component in the RINGS layer of PolderCast 28
4.2 Architecture diagram of VIZPUB 29
4.3 Snapshot of Gephi GUI with several important components annotated . 35
4.4 The Node Query Tool opens a panel (to the left) that can be used to inspect the attributes of a node, here it is used to inspect the subscription size of a high-degree node 36
4.5 The Statistics component is able to produce a HTML report which includes plots describing the distribution of values across the nodes . . . 38
4.6 The *Ranking Table* (to the left) in the Ranking Component may be used to sort nodes by attribute value. In this screenshot it is used to rank nodes by degree in descending order. 39
4.7 The Filter Component (to the right) enables users to strip away unwanted nodes and edges, in this screenshot it is used to display the sub-overlay for a specific topic by using a *regular expression*. 41
4.8 The Data Laboratory Component provides a table overview of metrics as well as the capability of exporting data. 42
4.9 Overlay structure of PolderCast at interval 0 44
4.10 Overlay structure of PolderCast at interval 250 44

4.11 Overlay structure of PolderCast at interval 500 44
4.12 Visualizations of average number of control messages sent and received per node in PolderCast 45
4.13 Visualization of publication message dissemination in PolderCast. . . . 47
4.14 Visualization of PolderCast after the end of dissemination 48
4.15 Visualization of duplicate publication messages received by each node in PolderCast 48
4.16 Visualizaton of the **overlay strucure** of PolderCast and Scribe after 1000 intervals 52
4.17 Visualization of **degree** in PolderCast and Scribe after 1000 intervals . . 53
4.18 Time series of number of nodes online during simulation produced by the *Statistics Component* in Gephi 55
4.19 Degree distribution of PolderCast and Scribe, produced by the *Statistics Component* in Gephi. 56
4.20 Time series of PolderCast and Scribe, produced by the *Statistics Component* in Gephi. 57
4.21 Visualization of a fully connected dissemination graph in PolderCast, which revealed a bug in the implementation of hit-ratio calculation . . . 59

5.1 Plots describing the difference of number of edges in Scribe when using different subscription workloads. 63
5.2 Avg. Directed Degrees of PolderCast and Scribe 66
5.3 Avg. Clustering Coefficient of PolderCast and Scribe 67
5.4 Avg. Betweenness Centrality of PolderCast and Scribe 67
5.5 Avg. Closeness Centrality of PolderCast and Scribe 68
5.6 Avg. Eccentricity Centrality of PolderCast and Scribe 68

List of Tables

3.1 Comparison of the different protocols and their overlay properties . . . 24
3.2 Comparison of the different protocols and their routing properties . . . 25

4.1 Reporter Interface Methods 30
4.2 Data Structure of a Publication Message 32
4.3 Calculated metric values for PolderCast and Scribe, calculated using the *Statistics Component* in Gephi. 50

Aknowledgements

First of all, my thanks goes to Roman Vitenberg and Vinay Setty the discussions, directions and feedback while working on this thesis. The assistance they provided before and during the DEBS conference was particularly helpful, as it was a hugely important factor in giving my thesis a much clearer goal and direction. Also, my thanks goes to Spyros Voulgaris who provided access to PeerNet, as well as technical support. A huge thanks goes to my family and friends for their support during the tough times, especially my parents, who supported me emotionally as well as financially by letting me move back in to my old room to hack away and write during the last few months of my thesis work. Many thanks to Robin, who let me use his couch when I needed to stay in Oslo a few days during the summer as well as showering me with his anti-academic rants which I found to be surprisingly motivating. My thanks also go to Anette and Christine, for their understanding and support. And lastly, I would like to give my thanks to to Jørgen for his unique perspective on life which always keeps my mood up.

Chapter 1

Introduction

The Publish-Subscribe (pub/sub) communication paradigm is receiving an increasing amount of attention from the research community, as it provides a loosely coupled and scalable interaction scheme suitable for large-scale distributed systems [11]. It has also shown to be a useful approach for several business applications such as financial data dissemination [31] and application integration [28]. Pub/sub systems have a large number of important performance metrics related to delivery latency, bandwidth consumption, communication overhead as well as computation overhead. Many of these aspects of pub/sub systems are known to be at stake with each other [17, 30], forcing researchers to face difficult design decisions when developing such systems in order to achieve a suitable balance in performance.

Overlays play a key role in the pub/sub system architectures, where many considerations need to be taken both with regards to overlay maintenance as well as structural overlay properties such as clustering and connectivity. These properties further add to the set of performance metrics which must be studied and weighed against each other by researchers and developers of pub/sub systems.

In this thesis we present VIZPUB, a tool we propose for visualizing the execution of any given pub/sub system step-by-step with respect to a set of performance metrics. Our tool enables logging local information at every node in the system under analysis, which is then collected at a single site for analysis. When all data has been collected, the tool will calculate global metrics at a system-wide scale. The resulting data is stored to disk in a single file using the .gexf [1] file format. This file is then interpreted by the Gephi Open Graph Viz tool [4] which enables the user to replay system execution as well as visualize overlay structure, dissemination paths of publications messages and various performance-critical metrics.

To the best of our knowledge, VIZPUB is the first tool of its kind. And although our main focus lies in the performance of pub/sub systems, the tool is generic in design, and can be used in order to visualize other types of overlay-based systems such as [21].

The major benefit of our tool lies in giving both researchers, developers as well

[1] GEXF (Graph Exchange XML Format) is a language for describing complex networks: http://gexf.net/format/

as students the ability to gain a deeper understanding of the properties of overlay-based systems, both in terms of their structure and their dissemination schemes. The tool is capable of providing the user with two different types of visualizations: (1) visualization of overlay structure and (2) visualization of publication message dissemination. Both types of visualization provides the user with the ability to replay the system execution as well as pausing and analyzing the overlay at different points in time. The visualizations mentioned above grants the user with an insight into the system which is helpful in order to determine protocol behaviour as well as potential weaknesses and anomalies of any given pub/sub protocol. In this thesis, we describe some of the practical experiences we had with the tool, which should serve as good examples of it utility.

Other benefits include the possibility of comparing different pub/sub systems visually. The user can run different systems with fixed parameters, driving the same workloads for publications and subscriptions, and then compare the performance and characteristics of the different systems at selected points in time. In Chapter 4 we include such a comparison of two different pub/sub systems.

As most of the interesting performance metrics across different pub/sub systems remains the same, we specify a generic list of performance metrics as well as a generic *reporter interface*. This interface is designed to provide the necessary data required to calculate these metrics and constitute the only part of the architecture which is system specific. This is an important point. The only work required from a researcher or developer who whish to take advantage of our tool is to implement this interface. The architecture of VIZPUB is designed to be modular and highly decoupled. The only contact point between the running system and VIZPUB is the reporter interface.

We also take advantage of the capabilities of VIZPUB in order to extend the evaluation performed in [30] on a specific set of structural overlay metrics. The advantage of using VIZPUB for this purpose lies in the fact that the Gephi framework grants us with several algorithms for calculating certain metrics for "free". Also, adding support for metrics which are not included in Gephi can be done by developing plugins for Gephi which can be distributed via its online marketplace [2]. This encourages sharing code between researchers, avoiding the individual researcher having to implement algorithms from scratch, which we believe would be of great benefit to the research community.

We presented a poster and held a live demo of VIZPUB at the ACM International Conference of Distributed Event Based Systems (DEBS), hosted in Mumbai in May 2014. The positive feedback from the researchers and students attending the conference reassured us that there is a demand for such a tool and that its potential usefulness is widely understood and appreciated. The value of our contribution was further demonstrated, as VIZPUB received the prize for best poster and demonstration. This reassures us that there is a need for such a tool and that VIZPUB may serve as a lasting contribution to the research community at large. In order to encourage contributions and further development of VIZPUB, we host our implementation code in a public

[2] Plugins for Gephi are available at the official Plugin Marketplace: https://marketplace.gephi.org/

repository [3].

1.1 Chapter Overview

The following is a brief overview of the chapters included in this thesis:

Chapter 2

In this chapter, we explain key concepts and describe the tools and technologies we use in the work presented in this thesis. This chapter is meant to serve as a brief overview of relevant topics and should prove useful in order to fully understand and appreciate the work presented in this thesis.

Chapter 3

We extend the mini-survey found in [30] with a set of additional pub/sub protocols. This chapter should be helpful in order to understand the different challenges researchers face when designing such systems. In particular it focuses on what trade-offs must be considered, and what are the advantages and drawbacks of certain design decisions.

Chapter 4

We describe VIZPUB, a tool we propose for visualizing the performance of overlay-based pub/sub systems. The chapter includes descriptions of the system architecture, as well as examples of visualizations. Also, we describe our experiences using the tool in order to further demonstrate its benefits both for researchers, developers as well as students.

Chapter 5

We use VIZPUB in order to expand the evaluation performed in [30] on a set of particular topology metrics. Our tool enable us to easily add these metrics to the evaluation, as the algorithm for calculating these properties of the overlay are included in the Gephi framework.

Chapter 6

The final chapter summarizes the contributions of this thesis. We discuss our results, as well as the future of VIZPUB. We also discuss what are the major challenges moving forward, and how the tool could be expanded. It is our hope that this tool will prove useful for the research community in the future.

[3] VIZPUB is hosted in a public repository: http://github.com/vizpub/vizpub

Chapter 2

Background

In this chapter, we describe some of the key concepts as well as important tools and technologies we use in the work presented in this thesis. The following sections are meant to serve as an brief overview of the different topics relevant to this thesis, which should be helpful in order to fully appreciate and understand the following chapters.

2.1 The Peer-to-Peer Network Architecture

In a Peer-to-Peer (P2P) network every node acts as both client and server. Every node contributes with its resources, including both storage space and processing power. The execution of the system is determined by a decentralized algorithm, which every node in the P2P network must follow. No node has global knowledge of the entire network, and no node acts as a single point of failure. This ensures a high degree of scalability in terms of number of queries or amount of data being processed, as every node is able to act as a server. Also, the P2P network architecture is highly resilient to churn, as each node independently needs to handle joins and leaves gracefully. This type of self-organization is one of the main characteristics of the P2P network architecture.

2.2 Overlays

The logical connections between participants in a P2P system lies on top of the physical connections on the network. This means that a one-hop connection between two peers might in reality consist of several hops between separate machines at the physical layer. The higher level connections between peers forms what is called an *overlay*, as they constitute a logical network at a higher abstraction level that facilitates routing, search and key-value storage. Typically, overlays are separated into two different types of networks: structured and unstructured. The former organizes nodes into structures such as trees or rings, while the latter aims to form an overlay which resembles a random graph. Structural overlays introduce overhead in terms of

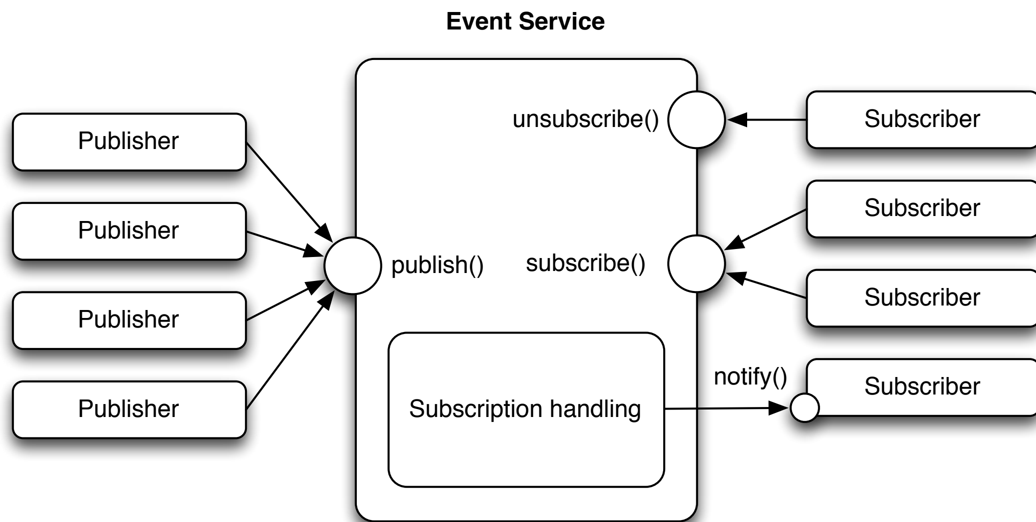


Figure 2.1: The basic architecture of a pub/sub system.

structural maintenance but are able to provide higher guarantees of correct message delivery than random overlays.

2.3 The Publish-Subscribe Communication Paradigm

Publish-Subscribe is a fully asynchronous, loosely coupled, highly scalable, event-based messaging pattern. There are three main system components in the pub/sub interaction scheme: the publishers, the subscribers and the event service. The publishers publish events, and the subscribers subscribe for events, while the event service handles managing both subscriptions and publications, as well as routing events to the subscribers. The basic architecture of a typical pub/sub system is outlined in Figure 2.1.

The event service functions as an intermediary between publishers and subscribers. It provides a level of indirection, as well as a service interface. Publishers are able to generate new events through the `publish` service call. It is now the responsibility of the event service to determine which subscribers are interested in receiving this event, and how to route the event to them. The subscribers register their interest through a `subscribe` service call. The event service will then store each subscribers interest in order to disseminate events correctly. The publishers are then able to cancel their subscriptions through a `unsubscribe` service call. No information is forwarded from subscribers to publishers or from publishers to subscribers.

The pub/sub paradigm provides a higher degree of decoupling than other traditional approaches, in general, there are three types of decoupling such a system

provides:

1. Space decoupling
2. Time decoupling
3. Synchronization decoupling

Space decoupling means there are no need for the publisher and subscribers to be know about each other. Subscriptions are handled by a third party. Time decoupling assures that events are delivered regardless of whether or not publishers and subscribers are online at the same time, while synchronization decoupling refers to the fact that neither publishers or subscribers are blocked when attempting to perform their operations. While many other approaches can provide the first two forms of decoupling, the main advantage of pub/sub is its fully asynchronous nature. Approaches such as tuple spaces or message queues cannot completely provide this synchronous decoupling, as messages are retrieved in a synchronous manner. This property is key to the suitability for pub/sub in large distributed system. [11]

2.3.1 Message Filtering in Pub/Sub

The subscription semantics of the pub/sub paradigm plays an important role in the performance and flexibility of the system as event messages are routed and managed based on topic or content. There are three distinct types of subscription schemes:

1. Topic-based
2. Type-based
3. Content-based

In Topic-based systems, events are split into topics which are usually represented as a string. In Type-based systems, events are filtered according to the structure of the data, which provides type safety at compile time. The Content-based approach filters events based on a global list of universal event attributes. This approach provides better expressiveness in terms of filtering out the relevant events, however, it also introduces more overhead with regards to handling subscriptions. The complex filtering algorithms in content-based approaches limit the scalability of such systems with regards to the number of subscriptions. Type-based share some similarities with content-based in the sense that the public members of the types together form a description of the content of the event. Although this ties the implementation of the pub/sub system closer to the programming language, it still suffers from the same drawbacks as content-based.

Topic-based offer less expressiveness than the other two subscription schemes, but better performance if the set of possible event properties is limited. Also, topic-based is more suited for dissemination and multicasting, as topics can be thought of as groups,

where subscribing to topic T can be equivalent to joining the group for that topic. This is a common approach taken by several proposed pub/sub systems [6, 12, 27, 30, 35]

Traditionally, reliable multicasting of data through deterministic dissemination has been the common approach. However, more recent implementations investigate the potentials of probabilistic protocols, which are more suited to the nature of decentralized systems and P2P. These protocols do not guarantee full reliability of delivery, but provides a high quantifiable *probability* that events are delivered to all subscribers.

2.4 The Gephi Open Graph Viz Platform

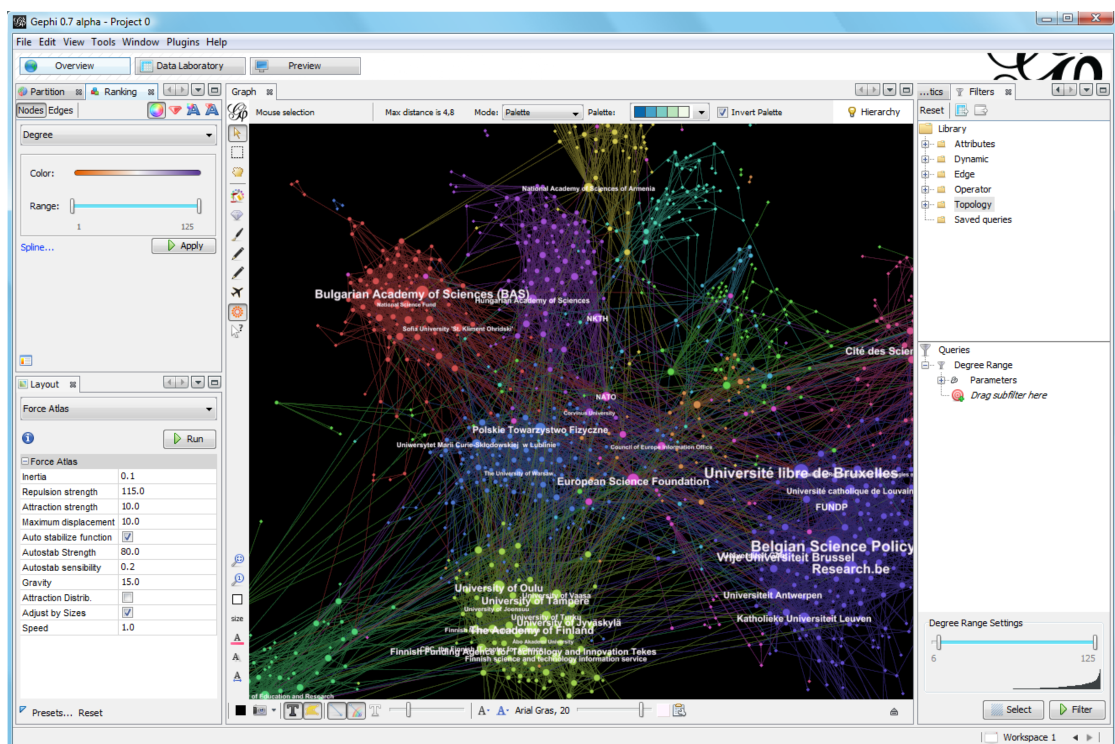


Figure 2.2: The Gephi Tool supports visualization of graphs through coloring and sizing the visual graph representation. It also enables adding labels to nodes and edges. In this screenshot, Gephi is used to detect and visualize communities.

Gephi [4] is an open source tool for exploring and visualizing all kinds of networks, including dynamic and hierarchical graphs. Described by the authors as "photoshop for graphs", Gephi enables the user to interact with the graph structure, as well as manipulate the colors and sizes of the visual graph representation in order to display graph properties in an intuitive way. Gephi aims to help researchers and data analysts in discovering patterns and revealing hidden properties of the graph in question, as

well as easily discovering errors in the dataset. Gephi also provides a set of statistical tools for measuring common metrics for Social Network Analysis (SNA) such as centrality, as well as metrics useful for general graph topology analysis such as degree, path length and clustering coefficient. Gephi is also useful in the emerging field of Dynamic Network Analysis (DNA) as it supports temporal graphs, giving the user the ability to filter the graph model according to a defined time interval. It also support playback of the graph evolution, as well as visualizing changes to graph data over time through size, color and text labels which can be applied to both nodes and edges.

We consider tools such as Gephi to be a valuable addition to the field of distributed systems research. Visual exploration of a dynamic network graph is a useful approach to evaluation and development of such systems, as some bugs are more easily spotted visually. For example, during our implementation work, it was trivial to visually confirm that some edges were missing from the graph visualization, leading to the discovery of a critical bug in the implementation code which would otherwise be difficult to spot. It is also worth to note that the different actors involved in the Gephi project has formed a legal entity in the form of The Gephi Consortium [9] in order to assure future development of this tool. This, along with the fact that there seems to be a growing community around this tool, gives us a certain degree of confidence that this tool is something well worth investing in, as the risk of it being discontinued or abandoned in the near future seems unlikely.

2.5 The Gephi Toolkit

In addition to the GUI-client, the authors of Gephi also provide an API through the Gephi Toolkit project. The toolkit packages essential modules from the GUI-client into a standard Java library which can be used by any stand-alone Java project by including it as a dependency. We take advantage of this toolkit in our implementation work, where it is mainly used to handle and store information collected from PeerNet simulations.

2.6 The GEXF File format

The GEXF (Graph Exchange XML Format) file format [10] is an effort by the Gephi Consortium to define a standard language describing complex network structures. Being developed by the same authors, the Gephi Tool is naturally fully compatible with this format, and is able to both import and export GEXF files. This is also the case with the Gephi Toolkit, as the module for handling such imports and exports are included in this toolkit.

The GEXF file format is able to describe a graph through its nodes and edges, as well as any data and dynamics associated with the graph. More specifically, the file format is able to describe node, edges and their associated attributes. Listing 2.6 provides an example of a minimal static GEXF file, describing nodes, edges and attributes of a graph.

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft" version="1.2">
    <graph mode="static" defaultedgetype="directed">
        <nodes>
            <node id="0" label="Hello" />
            <node id="1" label="Word" />
        </nodes>
        <edges>
            <edge id="0" source="0" target="1" />
        </edges>
    </graph>
</gexf>
```

Figure 2.3: A GEXF description of a minimal static graph

2.6.1 Dynamics

```
<gexf ...>
    ...
    <graph mode="dynamic" timeformat="date">
        <nodes>
            <node id="0">
                <spells>
                    <spell start="2009-01-01" end="2009-01-15"
                       />
                    <spell start="2009-01-30" end="2009-02-01"
                       />
                </spells>
            </node>
        </nodes>
    </graph>
</gexf>
```

Figure 2.4: Example of dynamic GEXF file using spells

One of the major advantages of this file format is its support for dynamic functionalities. Both nodes, edges and attributes may have a defined time interval where they exist. These lifetime intervals are described as *spells* if applied to nodes and edges, and as start and end XML-attributes if applied to node or edge attributes. The GEXF file in Listing 2.6.1 shows an example of a dynamic graph where spells are used in order to determine the lifetime of the nodes. The start and end times are by default encoded as floating point values, however, dates are also supported, as seen in

this example.

The support for dynamic graphs makes this file format an interesting option for storing simulation data, and in our implementation work we use this format extensively as part of our research effort.

2.7 Pub/Sub Protocols

We create visualizations and evaluate the performance for two different pub/sub protocols, namely PolderCast [30] and Scribe [32]. Both protocols provide a message dissemination scheme, where nodes are organized in a structured overlay. We briefly describe the relevant details regarding each protocol, as we later will provide visualizations of both the structure of these systems as well as their dissemination scheme.

2.7.1 PolderCast

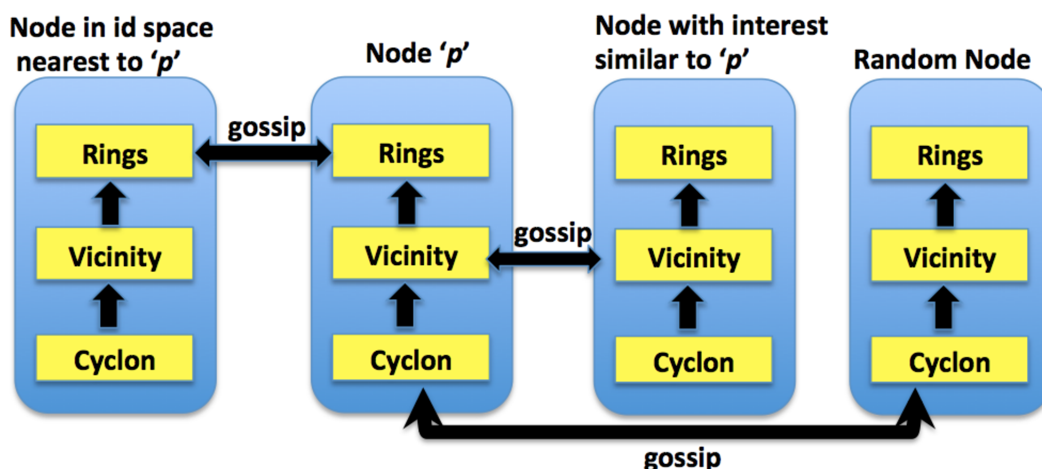


Figure 2.5: PolderCast consists of several decoupled components where a dissemination overlay is constructed at the RINGS layer

PolderCast [30] is a topic-based P2P pub/sub system which organizes nodes in a ring structure. The system architecture of PolderCast is highly modular, and includes three separate layers of overlay-based protocols. The CYCLON peer sampling service [32] is used in order to maintain connectivity across the whole set of subscribers, as well as providing the rings layer with uniform random links. The Vicinity module consist of the generic VICINITY protocol, which let nodes find neighbors based on a *proximity function*. This enables the construction of a RINGS where neighbors of nodes participating in the ring are based on the unique id of nodes, resulting in a ring sorted by node id. All the protocol modules rely on gossiping for structural maintenance. This

means that at periodic intervals, a node p will pair up with a neighbor q and exchange information regarding neighbors and subscriptions.

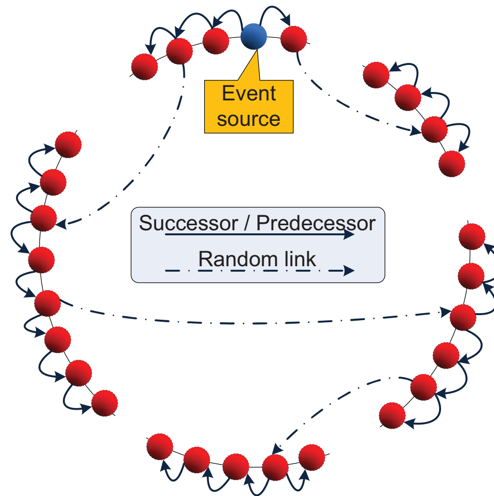


Figure 2.6: PolderCast utilizes a hybrid dissemination scheme when publishing messages, relying both on ring links as well as random links

PolderCast include a hybrid publication scheme using both ring neighbors and random links in order to boost dissemination, as well as increase resistance to ring partitions and churn. The dissemination algorithm is based on a configurable fanout f, and can be summarized in the following steps for each node:

1. If the message has already been received, discard it.
2. If the message was received from a ring neighbor, forward it to the other ring neighbor, as well as to $f-1$ random neighbors.
3. if the message was received from a random neighbor, forward it to both ring neighbors as well as to $f-2$ random neighbors

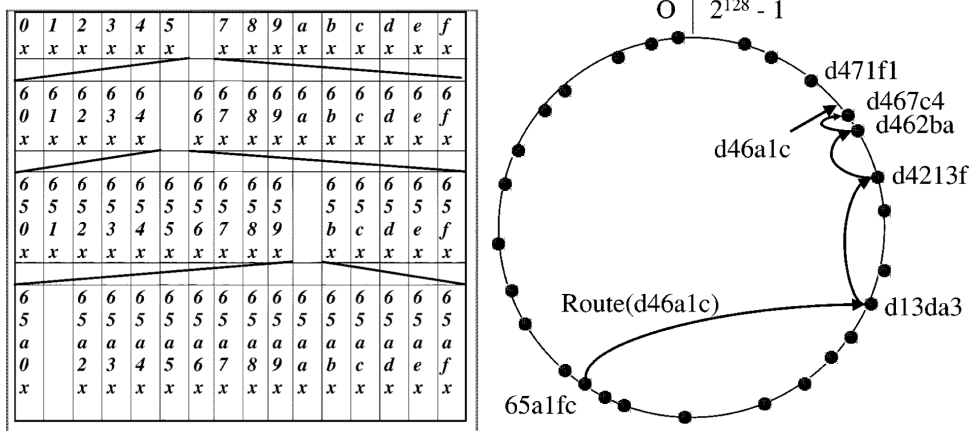


Figure 2.7: From left to right, the structure of a routing table in Pastry and an example of the Pastry routing scheme, where a message from node *65a1fc* with key *d46a1c* is delivered to the rendezvous node (figures borrowed from [6])

2.7.2 Scribe

Scribe [6] builds a dissemination overlay on top of Pastry [29], a distributed hash table (DHT) which provides other protocols with routing capabilities through an API. Scribe leverages these capabilities in order to provide group and membership management as well as message dissemination.

In Scribe, a tree structure is constructed for each group (i.e. topic). These structures are maintained by having nodes send heartbeat messages to its parent periodically. If a node suspects its parent is dead, it will use Pastry to find a new parent. When a node wants to publish a message, it sends the publication to the root node of the dissemination tree for that particular topic. This node is referred to as a *rendezvous node*, and is responsible for disseminating the publication to its children. The initial phase of the dissemination, where the message is routed to the rendezvous node, is handled by Pastry. Pastry uses a routing scheme based on unique node ids living in a circular namespace. Each node maintains a routing table of such ids, where each row n contains a sequence that match the id of the current node in the n first digits. When a node performs a lookup in the routing table, it will traverse the table like a tree. It will start by iterating through the entries in the first row until it finds a id sequence which matches the key on the very first digit, then it will lookup the row where the node ids all start with this digit, and iterate through the entries until it finds the sequence which is a match on both the first and the second digit. This will continue until an entry is found which shares a prefix with the lookup key which is longer than the current node id. If no such id is found, it will find an entry with a prefix with the same size as the current node id, but where the following digit is closer to the key. When the publication reaches the rendezvous node, this node will continue the dissemination by forwarding the publication to its children.

2.8 The PeerNet Simulator

PeerNet is an extended version of the popular *PeerSim simulator* [25]. More specifically, PeerNet adds a distributed simulation mode, where each node, or a subset of nodes, can be executed in separate processes. This means that unlike PeerSim, where execution of the protocols are performed sequentially, each node in PeerNet is able to execute in parallel. Also, in contrast with PeerSim, the execution of the system happens in real-time. The higher degree of parallelism and the real-time execution are two important factors which enable PeerNet to provide a more realistic environment for running experiments. It is also worth to mention that running in distributed mode is beneficial with regard to scalability in terms of number of nodes included in the simulation, as the memory and computational resources required to run the simulation can be distributed across several machines. This is an important factor as PeerNet is implemented in Java, which is a fairly memory intense language when dealing with large-scale systems.

While PeerNet is able to extend PeerSim with a distributed mode, it also provides a simulation mode which behaves exactly like simulations in PeerSim, where experiments are run locally in one single process. In our evaluation, we take advantage of both simulation modes in PeerNet. We update existing PeerSim implementations of PolderCast and Scribe for use with PeerNet, as well as implement a *reporter interface* for these protocols in order to make them compatible with our tool for visualizing and evaluating such systems.

2.9 Description of important metrics

In the following section, we describe some of the metrics visualized and analyzed in the following chapters. In general, the most common metrics important for performance in pub/sub can be divided in three categories: (1) structural overlay properties, (2) dissemination properties and (3) communication overhead due to overlay maintenance.

2.9.1 Structural overlay properties

Node Degree

> The degree of a node is determined by the number of connection to its neighbors. Degree can both be undirected and directed. Directed degree separates between in-degree and out-degree, where the former is a measure of the number of connection to this nodes from neighbors, while the latter is the number of outgoing connections from the particular node. A skewed distribution of directed edges would reveal an imbalance in the constructed overlay, or reveal vulnerable points in the system. Also, it is essential to understand how the system scales with regard to the number of topics a node is interested in. For example, if the number of edges increases linearly with the subscription size of a node, scalability suffers. A poor degree distribution affects load balancing, high degree nodes have

an increased likelihood of being overloaded, and might introduce bottlenecks in the system.

Topic diameter

This metric is a measure of the maximum number of hops between any two nodes that share interests, i.e. a measure of the diameter of a subgraph consisting only of nodes who registered their interest in the same topic. Having a low topic diameter is beneficial for disseminating events on a topic.

Clustering coefficient

This is the ratio of number of edges between neighbours of a node n over the total number of possible edges between them. In simpler terms, how many of a nodes neighbours are connected to each other. A high clustering coefficient would indicate that the network has a higher risk of partitioning, as well as a risk of having a higher number of redundant message deliveries.

Betweenness centrality

Betweenness is the number of times a particular node is found on the shortest path between two other nodes. A node with a big betweenness centrality may constitute both a vulnerable part of the graph as well as a bottleneck, as it might take part in a high number of event disseminations.

Closeness centrality

Closeness is a measure of how close the particular node is from every other node in the network. More specifically, it is the average distance in terms of shortest path length to every other node in the graph. Closeness is typically regarded as a measure of how long it will take to disseminate data from a node n to every other node in the network sequentially [26].

Eccentricity centrality

Eccentricity measures how far away a node n is from the node most distant from it in terms of path length. The maximum eccentricity is is the Graph Diameter, i.e. the longest path between any two nodes in the graph.

2.9.2 Dissemination properties

Hit-ratio during churn

Hit-ratio is the fraction of subscribers that received a publication message, over the total number of subscribers for that topic. It is essential to understand how the different systems respond to churn when disseminating events i.e. how the system deals with nodes both leaving and joining the network. If an overlay is robust, it should provide a high hit-ratio in the presence of realistic churn. Meaning that at very high percentage of subscribers receive the appropriate events.

Path Lengths

Dissemination path lengths is the number of nodes a publication message traverses before reaching its target subscriber i.e. the *number of hops* the publication messages makes from node to node. This metric is helpful in order to understand the efficiency of the dissemination algorithm.

Number of duplicate messages

Subscribers might receive the same publication message several times, depending on the method of dissemination being used. Epidemic dissemination schemes usually infer a higher number of duplicate messages. Redundant messaging incurs a overhead cost in terms of bandwidth as well as processing power at the receiving node.

2.9.3 Maintenance overhead

Number of control messages

Some systems rely on control messages in order to maintain the overlay topology. For example in Scribe, where the multicast tree structures are maintained with periodic heartbeat messages. The number of control messages a node sends and receives serve as an indicator of this overhead.

Control message bandwidth consumption

Control messages also constitute overhead in terms of bandwidth usage. Measuring the control message overhead in terms of number of bits sent and received from each node reveals the bandwidth cost, revealing any potential bottlenecks or overloaded nodes.

2.9.4 Using Test-Driven Development

Software Development Methodology is an active area of research which is in part driven by the business needs of the private sector[16]. One popular practice is so-called Test-Driven Development (TDD). The promoters of TDD claims it increases productivity and reduces the number of bugs and defects in the code significantly [5]. Research efforts performed at IBM [23] seems to lend credibility to these claims. However, the use of TDD is not prevalent in academia, and in [16] they recommend further research into the field in order to better determine its effects.

Using TDD means writing tests before writing any code. There are different types of test. *unit tests* targets small, independent pieces of code, typically methods within a single module or component, while *integration tests* aim to test code across such modules and components in order to determine how well they integrate with each other. In our work, we only took advantage of unit tests where we found it to be suitable. We could also have benefited from a suite of integration tests, as our implementation is heavily dependent on interoperating components, as well as file and network IO. However, writing these sort of tests would simply be too time consuming compared to writing smaller unit tests.

The TDD approach to software development is best described through the Red-Green-Refactor mantra, which is a central part of the TDD-philosophy. It can be described through the following steps:

Step 1: Write a test that fails. (Red)

Step 2: Make the test pass. (Green)

Step 3: Refactor the code while making sure the test still passes. (Refactor)

In our experience this routine has been helpful when working with our implementation code, as it enables us as developer to refactor with confidence achieving more maintainable code and a more thoughtful software design. Since we share our implementation code as open source, any tool or method that helps us improve the design and maintainability of our project is of great value to us. Using TDD forced us to think more deeply about what functionality to implement and how to structure and split the problem domain into smaller function points. We are confident that this practice decreased the amount of technical debt in our project, a problem we find to be commonplace in academia.

2.9.5 Sharing Code with the Community

We believe sharing our code is to the benefit of both VIZPUB and to the community. During our presentation at DEBS 2014, the interest in our tool was high, and we received a fair amount of requests for our implementation code. Due to PeerNet not being public yet, we have adapted our implementation of VIZPUB for PeerSim [25] using a publicly available implementation of Pastry [1]. When PeerNet is made public,

we will be able to host our original implementation. Sharing our code by hosting it publicly will hopefully allow the tool to grow from a prototype into mature product by allowing anyone to contribute with improvements and bug fixes. It is our hope that VIZPUB will be a lasting contribution to the research community, and hosting the code in a public repo[1] is a key part in ensuring a future for our tool.

[1] http://github.com/vizpub/vizpub

Chapter 3

Design Challenges in Topic-Based Pub/Sub

Designing decentralized topic-based pub/sub systems is a big research challenge due to a number of desired system properties which are in conflict with eachother. For example, making the overlay robust is difficult without introducing too many redundant edges in the network graph. Many approaches to topic-based pub/sub have been proposed the last decade [2, 6, 8, 27, 12, 22, 35, 36]. And each have made trade-offs in an attempt to balance the system properties against each other. In this chapter, we extend the mini-survey found in [30], were we include a number of additional protocols. Also, we will go more into detail regarding the design challenges found in topic-based pub/sub system by illustrating the charachteristics of these systems, as well as their shortcomings.

3.1 Desired System Properties

In order to provide correct and efficient delivery of notifications in a decentralized OSN using topic-based pub/sub, a high number of system properties are deemed desirable [30]. More specifically, these challenges include:

Correct delivery
: All notifications should be delivered to the correct recipient. Both false positives and false negatives should be avoided.

High hit-ratio during churn
: Notifications are delivered to a very high percentage of subscribers in the presence of churn. In the absence of churn, notifications should be delivered to all subscribers. This is similar to correct delivery except for not taking false positives into account.

Fast recovery
: The overlay should quickly recover from a period of churn. Nodes should be able

to both leave and join the network gracefully, and nodes who are dead should be properly handled by the system.

Low average node degree
The overlay nodes should have a low node degree as possible to achieve scalability with regards to number of topics. The degree distribution should be as even as possible, in order to achieve load balancing.

Topic connectivity
The routing of an event only includes the subscribers who registered their interest for the topic. This is also known as *relay-free routing*.

Scalability
The system should scale in terms of number of topics, number of nodes, number of topics a node is interested in and number of nodes interested in a topic.

Efficient dissemination
Event dissemination should have a low delay with little duplicate delivery, and the load of routing messages should be distributed fairly.

Low overlay maintenance cost
Managing the overlay topology should be as inexpensive as possible. Maintenance might include mending dissemination structures such as multicast trees when nodes fail, but also how to include joining nodes in the structure and allowing nodes to leave gracefully.

Designing a system with all of these properties presents a challenge, as several of the desired characteristics are fundamentally at odds with each other [17, 30]. Maintaining a low node degree makes it difficult to maintain *topic connectivity*, while avoiding duplicate message delivery conflicts with being robust in the presence of churn. There is also a trade-off in robustness and reliability depending on the approach taken to disseminating messages. Specialized overlays that build dissemination structures such as multicast trees provide fast and reliable message delivery with no duplication of messages. However, they are fragile and susceptible to churn. Epidemic dissemination on the other hand is more robust, but does not provide full reliability as it lacks deterministic delivery guarantees.

There is also a trade-off between the navigability of the overlay and the message overhead. Stanley Milgram famously demonstrated the *small-world phenomena* in [24] where he showed that any two participants in a network was likely to be connected through a low number of intermediaries. Taking this phenomena into consideration has proved to be a useful approach when constructing decentralised overlays, as they provide a highly navigable network due to the small average shortest path length. A popular approach is to create one or more long jump links between nodes to provide better routing capabilities. More specifically, these links are usually created by utilizing a distance function in the name space, where the probability of creating a link increases

with the distance between them. The subscription interest of such nodes are usually not taken into consideration when creating such links. Consequently, the message overhead in the system is increased as more relays are introduced in the overlay.

Many existing systems suffer from shortcomings that originate from wanting to include a high number of desired properties described above. This motivates further research into these systems and how they compare in terms of promoting these desirable properties.

3.2 Handling Trade-offs

Designers of existing approaches have been facing the challenges of handling the trade-offs discussed in the previous section. One of these challenges is building both a reliable and robust overlay. A naive approach to this problem would be to create a separate overlay for each topic as in TERA [**Baldoni:2007**]. However, this approach suffers from poor scalability as the number of nodes and topics increase. Another approach would be to structure the overlay by creating a spanning tree per topic as seen in Scribe [6], Bayeux [36] and Magnet [12]. However, these structures are conceptually fragile in the presence of churn, requiring mechanisms for mending the structure when nodes fail. This increases the overhead of overlay maintenance. Also, the root node of the spanning tree represents a single point of failure as well as a bottleneck in the system. This is especially true for popular topics where all events must travel through the root node. In Scribe, the root node is used as a *rendezvous* point for topics by using the routing capabilities of the underlying Pastry DHT [29]. Such dedicated nodes represent a single point of failure in addition to being detrimental to load balancing and scalability.

Minimizing the average node degree while simultaneously achieving a *topic-connected overlay* (TCO) is another difficult trade-off to consider. Topic-connectivity is achieved when no other than the nodes who registered their interest in a topic takes part in routing events for that topic. The desired goal with regards to topic-connectivity is not only to avoid routing events through uninterested nodes, but also to minimize the node degree by reusing links for several topics. This approach achieves better scalability with regards to the number of topics in the system. Also, it decreases the message overhead incurred by both event dissemination and overlay maintenance mechanisms such as heartbeat messages. In addition to this, keeping an overlay topic-connected simplifies the message routing mechanism as no designated relay or gateway node needs to be implemented in the protocol such as in Scribe and Vitis.

ElastO [7] propose an interesting approach to overlay construction, where the goal is to construct a TCO while maintaining a low node degree. The construction of the overlay is performed by a centralized component which requires global knowledge, while the maintenance of the overlay is performed in a distributed manner in response to churn events. By using a centralized algorithm for overlay construction, ElastO is able to provide a more optimal TCO than decentralized solutions, while still maintaining the high performance of a decentralized repair mechanism to handle node

departure or arrival.

With regards to the reuse of links for several topics, an observed correlation [20] between subscription sets in practical workloads is useful to consider when constructing overlays. This observation is exploited in Poldercast in order to decrease the number of links to maintain. Also, it is used as a basis for overlay construction in both StaN [22] and SpiderCast [8]. However, these two protocols only provide a probabilistic guarantee that the resulting overlay will be fully topic-connected. In contrast, PolderCast claim deterministic guarantees of providing a TCO. However, this relies on two factors: (1) that there is no churn in the system, and (2) that the underlying Cyclon [32] protocol, which is used for peer sampling in PolderCast, can guarantee a connected overlay. Consequently, the deterministic guarantees of PolderCast could be questioned. daMulticast [2] on the other hand provide a deterministic guarantee of topic-connectivity through quite a different approach of overlay construction. More specifically, daMulticast constructs a topic hierarchy, where events are disseminated through gossiping each level of this hierarchy in a bottom-up approach.

As mentioned in section 2, several protocols attempts to create an overlay that exhibits *small-world properties*. In Vitis, the subcluster together with the relay paths form an overlay similar to a small-world networks, which decreases the routing delay, but includes uninterested gateway and relay nodes. In Magnet, small-world properties are provided by the underlying Oscar DHT [13] which cluster similar nodes together.

3.2.1 Overlay construction

There are several different approaches to overlay construction. Structured approaches such as dissemination trees have already been mentioned, but there are also unstructured approaches. In Quasar [35], a novel approach to event dissemination using random walks removes the need for a structured overlay. There are also hybrid approaches to structuring overlays such as in ElastO [7] and Vitis [27]. In ElastO, the bootstrapping of the graph is performed by a centralized entity, while in Vitis nodes with similar interests are clustered together. A topic in Vitis might consist of several subclusters which are connected to each other through relay paths. This creates an overlay that is similar to dissemination trees, but where single nodes have been replaced with clusters of nodes. However, the drawbacks are still similar to the ones found in systems relying on multicast trees, as it relies on designated gateway nodes within subclusters communicating with rendezvous nodes along the relay path. In Poldercast [30], a structured ring per topic is used in combination with a form of epidemic dissemination that resembles gossiping. Publishers are themselves part of the ring of the topic they publish, and the structures attempt to combined into themselves into a single overlay through random links. Such an hybrid approach is an attempt at balancing the reliability of a structured overlay with the robustness of epidemic dissemination. When it comes to node degree however, Poldercast might introduce hotspots in the system as the distribution of random links might be skewed.

In Magnet, the aforementioned subscription correlation is used to build dissemin-

ation trees such as the ones seen in Scribe and Bayeux. As mentioned these structures are not ideal in dynamic systems as they require maintenance. However, tree structures do have an advantage when it comes communication overhead, as they in theory avoid any duplicate messages. In practice however, duplicate messages might occur if a node that is part of a tree is part of an underlying routing path to the root node. For example, in Scribe, notifications are routed to the root node of a topic tree using an underlying DHT. If a child node belonging to this tree is part of the routing path, it will receive the same message twice. Once while routing to the root, and once again from its parent after the publication message has reached the root node. However, the number of duplicates should still be lower than what is seen in systems relying on epidemic dissemination schemes, such as daMulticast and PolderCast. In such systems, the number of duplicate messages are indeed higher, however, the increased number of control messages also has the benefit of making these systems more resilient to churn. Furthermore, there is usually an adjustable *fanout parameter* in such systems which can be manipulated in order to control the number of messages that are forwarded by a node. Thus, there is some control over the amount of communication overhead in these systems. Both PolderCast and daMulticast include such a fanout parameter. Also, it bears mentioning that structured overlays include communication overhead in the form of structure maintenance and structure mending in the case of both node failure as well as nodes leaving and joining the network. Control messages such as heartbeats are commonplace in such protocols e.g. in Scribe where the each non-leaf node in the multicast tree periodically sends heartbeats to its children. This increases bandwidth consumption and adds a higher communication overhead compared to unstructured overlays where such maintenance usually is not required.

Node degree is another important issue to consider when designing overlays. A low average node degree increases scalability as topics and number of nodes in the system increases. In protocols who rely on an underlying DHT, node degree is usually either constant or a logarithmic function of the total number of nodes in the graph. Such is the case in Bayeux which relies on the Tapestry DHT [**tapestry**], or Scribe which relies on the Pastry DTH [29]. Other implementations might have a constant node degree, which is the case in Vitis which might result in the separation of a topic into subclusters. Some extreme examples include TERA [**Baldoni:2007**] and daMulticast [33] which has a node degree that grows in the order of the number of topics the node has subscribed to. In the worst case scenario, this is also the case in PolderCast, as maintaining a low node degree depends on the degree of correlation in the subscription sets. Indeed, when using workloads from Facebook, the node degree in PolderCast grows almost linearly with subscription size, as shown in [30]. This suggests a scalability issue in a scenario where the subscription correlation is weak.

Table 3.1 provides an overview over several different state-of-the-art protocols, comparing their different system properties such as topic connectivity and whether or not it takes advantage of subscription correlation. Note that PolderCast has received the benefit of doubt in this table, and have been marked as providing a deterministic guarantee of topic-connectivity. Magnet has a different approach to the spanning tree

Protocol	Overlay	Structures?	TCO?	Central nodes*	sub. corr.?	Node degree
Scribe [6]	Structured	Trees	✗	RV	✗	$O(\log \lvert\mathcal{V}\rvert)$
Magnet [12]	Structured	Trees	✗	Relays	✓	$O(1)$
Bayeux [36]	Structured	Trees	✗	RV	✗	$O(\log \lvert\mathcal{V}\rvert)$
Vitis [27]	Hybrid	Trees	✗	RV&GW	✗	$O(1)$
StaN [22]	Unstructured	None	prob.	None	✓	$O(\lvert\mathcal{T}_v\rvert)$
SpiderCast [8]	Unstructured	None	prob.	WB	✓	$O(K \cdot (\lvert\mathcal{T}_v\rvert))$
daMulticast [2]	Unstructured	None	det.	None	✗	$\Theta(\lvert\mathcal{T}_v\rvert)$
Quasar [35]	Unstructured	None	✗	None	✗	Unknown
PolderCast [30]	Hybrid	Rings	det.	None	✓	$O(\lvert\mathcal{T}_v\rvert)$
ElastO [7]	Structured	Ring	det.	None	✗	$O(\rho \log \lvert\mathcal{V}\rvert \lvert\mathcal{T}\rvert)$

*RV: Rendezvous GW: Gateway WB: Weak bridge

Table 3.1: Comparison of the different protocols and their overlay properties

structures, where messages are disseminated bottom-up. This means that the root node is not a rendezvous node according to the traditional definition [3], but it is still conceptually a single point of failure as it is responsible for propagating messages back down the tree when it receives a message. In SpiderCast, there is a possibility of the overlay forming into a pattern of highly connected clusters inter-connected through a small number of links which we refer to as *weak bridges*. The node degree in SpiderCast also relies on the *K-coverage* parameter of the system, where, for each topic, a node attempts to connect to K neighbours who share the same interest. Protocols who rely on a underlying DHT typically have a node degree which grows logarithmically with the number of nodes in the system. The exception is Magnet which leverages a DHT providing small-world properties [13], creating a fixed node degree that is independent of both subscription size and number of topics [36]. Note that the node degree in Quasar is omitted from the table, as it is dependent on the implementation of the bloom filters used to represent the neighbours. In ElastO, ρ is a system parameter which balances between average and maximum node degrees when choosing new edges to recover from churn.

3.3 Event dissemination

In terms of event dissemination, it should be mentioned that some of the systems described earlier do not concern themselves with this aspect, and focus only on the construction and maintenance of the overlay itself. More specifically, this includes SpiderCast, StaN and ElastO. Thus, any discussion regarding dissemination technique or routing performance will be irrelevant for these systems. For other systems however, a comparison of techniques is in order.

As mentioned, the dissemination in systems relying on multicast trees removes much of the message duplication and usually offers on average dissemination of events in logarithmic time as is the case in Scribe, Magnet and Bayeux. In Vitis, event dissemination is performed by flooding inside the subcluster, while simultaneously

Protocol	High hit-ratio during churn	100% hit-ratio in absence of churn	Message Delay	Avg. Duplication Factor
Scribe [6]	✗	✓	$O(\log \|\mathcal{V}\|)$	None
Magnet [12]	Unknown	Unknown	$O(\log \|\mathcal{V}\|)$	None
Bayeux [36]	Unknown	Unknown	$O(\log \|\mathcal{V}\|)$	None
Vitis [27]	✓	✓	$O(\log^2 \|\mathcal{V}\|/k)$	Scoped flooding
daMulticast [2]	✓	✗	$O(\log \|\mathcal{V}\|)$	Gossiping
Quasar [35]	✓	✗	Unknown	Random Walk
PolderCast [30]	✓	✓	$O(\log \|\mathcal{V}_t\|)$	$\leq fanout(f)$

*RV: Rendezvous. GW: Gateway. WB: Weak bridge.

Table 3.2: Comparison of the different protocols and their routing properties

forwarding the event to other subclusters if needed. As mentioned, gossiping is the main approach in both daMulticast and PolderCast. Gossiping usually implies an exponential dissemination speed, however, there might be other implementation specific factors in play which inhibits this property of gossiping. As an example, in PolderCast, skewed random link distribution might be detrimental to the speed of the gossiping protocol. The authors of Quasar [35] propose quite a different approach to event dissemination, as routing is performed by having nodes install routing vectors in nearby overlay neighbours. Messages are disseminated through random walks, which are directed towards the subscribers when passing through a node with the relevant routing information. This approach is likely to be highly robust against churn. However, as observed in [35] the hit ratio stagnates at 97% in a static system. This is due to a phenomenon where some group members might be obscured by other members who absorb messages from all directions. The authors suggest a solution by having node periodically pull information from other nodes, but this introduces more overhead in terms of network traffic and data processing.

Table 3.2 describes the routing properties of the protocols discussed in this section. Note that protocols relying on a DHT usually have an expected delay which is logarithmic or squared logarithmic with the total number of nodes in the system. In Vitis, the underlying DHT provide squared logarithmic routing complexity with the total number of nodes divided by k, the number of long-range neighbours. To the best of our knowledge, there is no evaluation of the hit-ratio of Magnet or Bayeux, which is the reason of these being marked as unknown. Message delay is defined as the expected path length of a dissemination message in terms of number of hops. Systems relying on an underlying DHT usually provides routing performance which is logarithmic to the number of nodes in the system. Magnet differs in its approach to routing, as it relies on random walks with associated *time-to-live* values. However, as described in [12] this value is usually set to the logarithm of the total number of nodes in the system. The average duplication factor describes the message overhead of the system, where gossiping is usually dependent on a fanout system parameter. The novel approach in Quasar means message overhead is dependent on the number of parallel random walks initiated by a node. As described earlier, protocols which create specialized dissemination structures, in these cases spanning trees, have in theory no message duplication.

3.4 Chapter Summary

In this chapter, we extend the mini-survey found in [30]. We describe several different topic-based pub/sub protocols and their properties. In particular, we focus on the several trade-offs that researchers need to consider when designing such systems. We then describe different system and how they relate to these trade-offs by describing their properties and how it affects their performance in terms of overlay construction and maintenance, as well as event dissemination.

Chapter 4

Visualizing Performance in Overlay-based Pub/Sub Systems

In this chapter we describe VIZPUB [18], a tool we propose for visualizing the performance of overlay-based Pub/Sub Systems. In addition to describing the tool and its system architecture, we present several examples of visualizations produced by using our tool. Also, we use VIZPUB in order to compare PolderCast and Scribe visually, both using visualizations as well as using VIZPUB in order to produce certain plots. We also discuss the benefits of using visualizations when studying and analyzing pub/sub systems, where we share several experiences using our proposed tool.

We presented a poster and held a live demonstration of VIZPUB at the ACM International Conference of Distributed Event Based Systems (DEBS), held in Mumbai in May 2014, where it was awarded the price for best poster and demo. Also, our implementation of VIZPUB is open source, and available in a public repository.[1] It is our hope that our tool will be of benefit to the community, and aid researchers in further development and study of overlay-based systems.

To the best of our knowledge, VIZPUB is the first tool of its kind. The tool is able to visualize the execution of any given distributed pub/sub system step by step with respect to a set of performance metrics. Each node in the system records relevant data at selected intervals during system execution. Our tool is then able to pull this data to a single site, and compute various metrics at a system-wide scale. The collected data from each interval is then collated into a single .gexf file, which is interpreted by the *Visualization Unit* which enable replay of system execution offline.

Our tool supports two different types of visualizations: (1) visualization of the overlay structure and how it evolves over time and (2) visualization of publication message dissemination, where directed edges represent the dissemination path. We provide examples of both types of visualizations later in this chapter.

There are several benefits to using a tool such as VIZPUB. It enables researchers and developers to gain a deeper insight into the overlay structure as well as the publication

[1] VIZPUB is open source and hosted at http://github.com/vizpub/vizpub

process. It also has major benefits as an educational tool, as it provides students with a visual representation of both the structural evolution of the system under study, as well as a step-by-step animation of publication message disseminations. This is useful in order to engage students, and to facilitate deeper insight into the different pub/sub systems and their dissemination schemes. Such an insight is also useful in order to identify potential weaknesses or deployment anomalies of any given pub/sub system. While developing our tool, we encountered many scenarios where VIZPUB demonstrated its usefulness. For example, when experimenting with visualizations of PolderCast, we could immediately verify that three nodes were disconnected at the RINGS layer, as seen in Figure 4.1. Using our tool, we were able to verify that this was caused by an artefact in the input workload where the three nodes had no overlapping interest with any other node in the overlay. We were then able to confirm that the nodes were connected at the CYCLON layer. We are not aware of any other tool or framework that would allow such easy detection and validation of system behaviour.

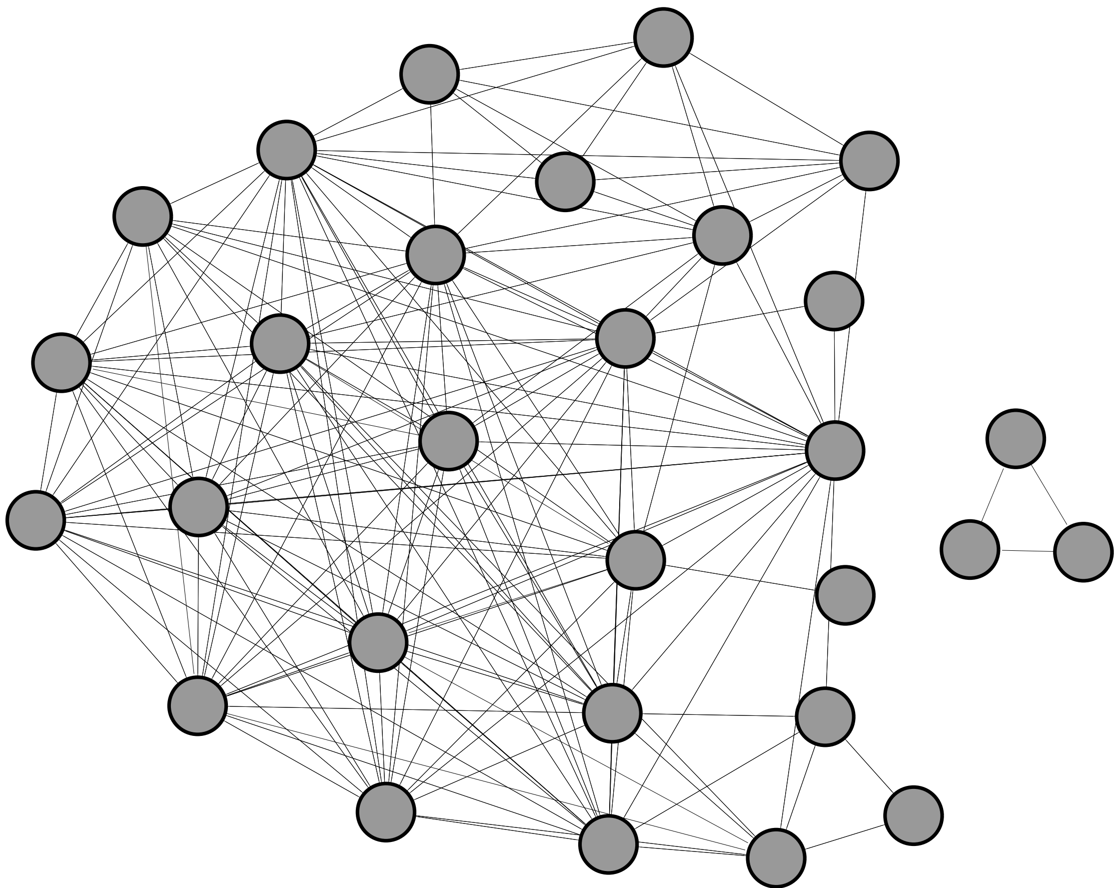


Figure 4.1: Visualization of disconnected component in the RINGS layer of PolderCast

Another interesting use case for our tool is comparing different pub/sub systems

visually. Users may run the different systems using the same system parameters as well as using the same workloads for subscriptions and publications in order to replay the execution and then perform a visual comparison. We include such a comparison in this chapter, were we compare PolderCast and Scribe, in order to see what we can learn from such a visual analysis.

4.1 System Architecture

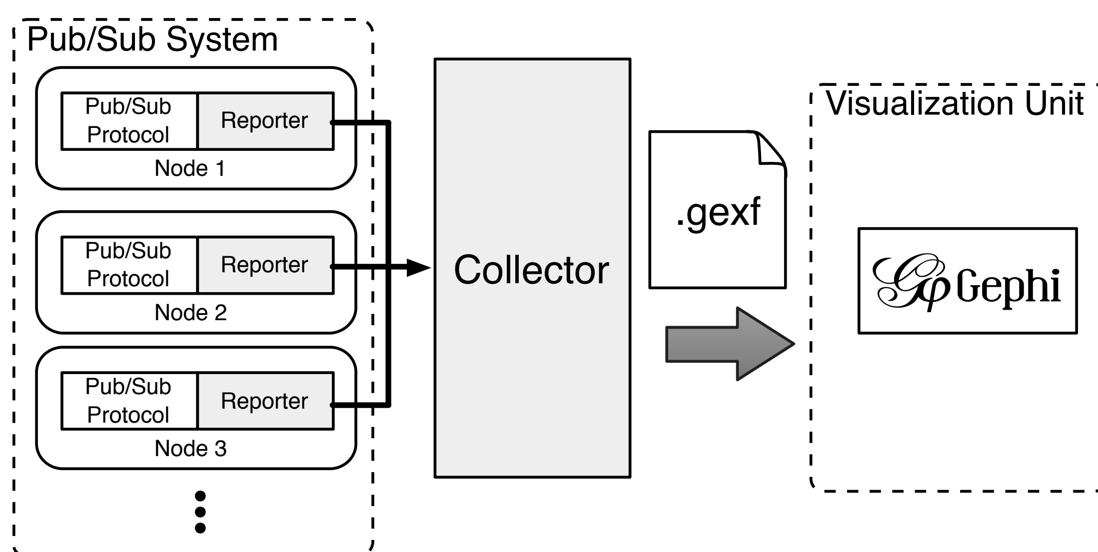


Figure 4.2: Architecture diagram of VIZPUB

The architecture of VIZPUB consists of three main components, all depicted in Figure 4.2: (1) *Reporter*, (2) *Collector* and (3) *Visualization Unit*. The arrows seen in Figure 4.2 depicts the flow of data in the architecture. Each node in the executing system consists of a pub/sub protocol as well as a "Reporter". The Reporter is the entity responsible for providing the raw information required to compute various performance metrics from the individual nodes participating in the overlay. This information is pulled at regular intervals to a central site by the "Collector", which stores the information as a single file in the .gexf format. This file is then interpreted by the "Visualization Unit", which consists of a single machine running the *Gephi Open Graph Viz tool* [4]. The Collector is designed to perform the collection of data while in online mode, while the computation, aggregation and derivation of various metrics is performed in offline mode. The Visualization Unit always operate in offline mode, as all the individual reports pulled from the different reporters needs to be collated into a single .gexf file before it can be interpreted in Gephi for playback and visualization. The processing of the individual reports into this final report happens *after* system execution.

VIZPUB supports pub/sub systems that are deployed as real distributed systems, as well as systems that are deployed in simulation mode. This is due to the highly modular system architecture, with strict separation of concerns, where the operation of both the Collector and the Visualization Unit is kept separate from the reporting.

VIZPUB is also designed to be a generic tool, where the only system specific part of the architecture is the *reporter interface* outlined in Table 4.1. Any researcher or developer who wants to use our framework only needs to provide an implementation of this interface, which enables the Collector to retrieve the relevant data from each individual node by calling the methods of this interface. The time points for collection are configurable, and all the information collected from the Reporter can be thought of as a representation of the change of system state from the current time point to the previous time point with regards to the various performance metrics. More specifically: the change of system state that occurred in the *interval* between two time points. Thus, we call these time points *reporting intervals*. The length of the intervals are configurable, which provides the user with control over the granularity of data collection, and thus the granularity of the step-by-step replay of system execution performed in the Visualisation Unit. For example, if running simulation using PeerNet, the user may determine whether or not the reporting intervals should encompass several simulation cycles. Or, in a real distributed pub/sub deployment scenario, the user can determine the time delay between every reporter interval.

4.1.1 Reporter

The Reporter is responsible of providing the relevant data necessary in order to calculate the desired performance metrics. In order to do so, we specify a *reporter interface* which must be implemented at each individual node participating in the pub/sub overlay. This interface enables each node to log certain system parameters using its local knowledge at each reporting interval. This local information is then pulled by the Collector at the end of each interval by invoking the reporter interface. The available interface method calls and what data they return is described in Table 4.1.

Method Name	Returns
`long reportId()`	The unique id of this node
`long[] reportNeighborIds()`	The unique ids of this node's neighbors
`long[] reportTopics()`	List of topic ids this node subscribes to
`long reportControlMsgsReceived()`	Number of overlay control messages received
`long reportControlMsgsSent()`	Number of overlay control messages sent
`long reportControlBytesReceived()`	Number of overlay control bytes received
`long reportControlBytesSent()`	Number of overlay control bytes sent
`PubMessage[] reportPubMsgsReceived()`	Reports list of publication messages received
`PubMessage[] reportPubMsgsSent()`	Reports list of publication messages was sent

Table 4.1: Reporter Interface Methods

It is easy to see how certain performance metrics can be derived from the methods listed in Table 4.1. The structural properties of the overlay such as *degree*, *diameter* and *clustering coefficient* can all be derived by reconstructing the overlay topology and running the specific metric using the *Statistics Component* in Gephi. The reconstruction of the overlay can be achieved by collecting the information returned by the two first methods listed in Table 4.1, namely `reportId()` and `reportNeighborIds()`. For example, in our reporter interface implementation for the RINGS layer in PolderCast, each node returns its own id as well as the ids of both ring neighbors and random neighbors. After this information is pulled, the Collector is able to derive a graph structure where it first builds nodes based on the information returned by `reportId()` and then draw directed edges between these nodes based on the data returned from `reportNeighborIds()`. What topics each node subscribe to is also useful in order to derive and visualize metrics such as *Topic Diameter* and *Subscription Size*. The Collector is able to pull information regarding topic subscriptions through the `reportTopics()` method call. Each node will return a set of topic ids, and the Collector is able to use this information to attribute topics to nodes as well as edges. In order to add topics to edges, the Collector simply iterates through the topic id list of each node, and looks for a neighbor who share a subscription to the same particular topic: a *topic neighbor*. If a topic neighbor of a node is found, the topic id is added as an attribute to the edge connecting them. Applying topic attributes to nodes and edges, provides the Visualization Unit (i.e. Gephi) with the ability to strip away nodes and edges that does not belong to a particular topic, thereby enabling calculation of topic diameter.

The dissemination properties of a given pub/sub system such as *hit ratio*, *path lengths*, and *number of duplicate publication messages sent/received* can be derived by having each node provide a list of publication messages sent and received. In order to calculate these dissemination metrics, the publication needs to have a particular structure. This structure is described in Table 4.2. For example, in order to calculate hit-ratio for a specific topic, we need to divide the number of subscribers of that topic who actually received the message with the total number of topic subscribers. We already know which nodes subscribe to a particular topic through the `reportTopics()` method call, and the list of publication messages received by a node can be retrieved through `reportPubMsgsReceived()`. Path length of a message being published on a particular topic from a particular node may be calculated in a similar fashion, as different copies of a publication message reported from different nodes can be ordered based on their timestamp values.

The number of duplicate publication messages sent and received by each node is available through the `reportControlMsgsSent()` and `reportControlMsgsReceived()` respectively, while the communication overhead incurred by control messages in terms of bandwidth consumption can be derived by the `reportControlBytesSent()` and `reportControlBytesReceived()` method calls.

The structure of the publication messages outlined in Table 4.2 also allows for visualizing the paths of publication messages. As mentioned, this is one of the two types of visualizations the VIZPUB is able to produce (where the other type is the

Message item	Description
`long MsgId`	Unique id of the this message
`long TopicId`	Topic id for which this message was generated
`long SourceId`	Id of the previous hop node
`long DestinationIds[]`	Ids of the next hop nodes
`long OriginalSenderId`	Node id of the message source
`long TimeStamp`	Timestamp of the message sent/received

Table 4.2: Data Structure of a Publication Message

overlay structure). The Collector will output a separate .gexf file for this type of visualization. When creating such a visualization, the Collector will look at the topic id of the message, and only include the nodes interested in the particular topic. The Collector will then iterate through the messages sent and received by each node. By analyzing the messages further, the Collector is able to create directed edges between the nodes which represent the path of the publication message. The edges are dynamic, i.e. they include a "Time Interval" attribute, enabling a step-by-step animation, where edges appear as the animation is played back in Gephi. These edges trace the path of the publication hop-by-hop. This enables researchers, developers as well as students to analyze publication dissemination schemes visually, which enhances their ability to understand, study and debug dissemination algorithms.

In addition to being able to configure the reporting intervals, users of VIZPUB may choose to only report partial information. For example users may choose to only report structural information such as node ids and neighbor ids, or only dissemination specific data such as publication messages sent and received. This flexibility is useful if only a few aspects of system performance require analysis.

4.1.2 Collector

The Collector is the component responsible for pulling information from the nodes at every reporting interval. It is also responsible for aggregating and calculating certain *custom metrics*. By custom, we mean any metric that is not included in the *Statistics Component* of Gephi, described in Section 4.1.3. These metrics are usually related to dissemination and include *hit-ratio*, *duplicate publication messages received* and *path lengths*. Metrics related to overlay structure can be calculated in Gephi. These metrics include *degree*, *clustering coefficient*, *diameter* and *centralities*. There is one topology metric which needs to be calculated in the Collector, namely *Topic Diameter*. In order to calculate topic diameter, the graph needs to be filtered down to a subgraph which only includes nodes and edges for a given topic, then it must calculate the path length for every such subgraph. Doing this manually using the Gephi GUI-client would be a time consuming and error-prone task. Therefore, the Collector takes advantage of the *Gephi Toolkit* in order to automate this task.

The Collector supports both what the Gephi community refers to as *static* and *dynamic* metrics. This is also referred to in literature and in [18] as *instantaneous* and *aggregated* metrics. In this thesis, we will refer to them as static and dynamic, in order to be consistent with the terminology used by the Gephi community. In short, static metrics pertains to a specific point in time, while dynamic metrics are based on historical values. The Statistics Component in Gephi includes support for calculating both type of metrics, but dynamic metrics only include degree and clustering coefficient, while the Collector is able to compute dynamic metrics for all properties such as centralities, hit-ratio and number of control messages sent and received.

Aggregation of data is performed by serializing each individual report received from the reporters into temporary files which are stored on disk. The Collector will then iterate through these files and output a final report in the .gexf file format. This file can then be used to create a range of visualizations, using the *Visualization Unit* described in Section 4.1.3. The .gexf file can be thought of as a sort of "replay" file, which can be transferred to any other computer running the *Visualization Unit*. Storing the execution of a pub/sub-system in a portable file grants researchers and developers a high degree of flexibility which we believe is of great value. Users of VIZPUB may share this file between them, educators may make such files available for students in order to study different pub/sub protocols, and developers may share such files in order to aid each other when debugging.

While collection of reporter data is done in online mode, aggregation is performed in offline mode. The offline aggregation of data prevents the Collector from acting as a bottleneck. Indeed, the collection and aggregation of data is highly decoupled from the execution of the pub/sub protocol itself. As an alternative, the Reporters are also able to log reports locally, and push them to the Collector at the end of pub/sub execution.

The Collector will use the information pulled from the reporters in order to apply attributes to nodes and edges. These attributes form the basis of node labels and colors when visualized in Gephi. For example, the number of control messages sent is a node attribute which can be represented as a numeric label on nodes. Also Gephi will inspect all nodes for their "control messages sent" attribute value, and determine the maximum and minimum value. These value form a value range which can be used to color the nodes on a gradient. For example, the closer a nodes value is to the maximum, the deeper the color of the particular node. Examples of visualizations using color can be seen in Section 4.2.

Some of the attributes are indirectly derived from the reported information, such as the *subscription size* of nodes, which is derived from the length of the collection returned by `reportTopics()`. What attributes to apply is configurable. The only edge attributes supported by the Collector is the *Topics* attribute. This is derived by determining the set of subscribers of a topic and analyzing the edges between them. If two nodes have an edge between them, they are topic neighbors. The list of node attributes supported is far more extensive and includes:

- Control messages sent

- Control messages received
- Control messages sent in kilobytes
- Control messages received in kilobytes
- List of topics subscribed to
- Number of topics subscribed to (i.e. *Subscription Size*)
- Number of duplicate publication messages received

These attributes describe data pertaining to individual nodes. However, the Collector will also calculate data which is global to the entire overlay. For example, the Collector is able to calculate averages such as average control messages sent per interval. The Collector does this by iterating through the interval range, and sum the numeric attribute value of each node that exists in this interval before dividing this number with the number of existing nodes, i.e. it calculates the *arithmetic mean* of each interval. The resulting averages are applied to all nodes as labels, even though they represent a global value, i.e. it is not a value specific to the particular node. This is a workaround, as Gephi does not support displaying graph attributes. The global attributes calculated by the Collector include:

- Hit-Ratio
- Average Number of Control Messages Sent
- Average Number of Control Messages Received
- Average Number of Control Messages Sent in kilobytes
- Average Number of Control Messages Received in kilobytes

The Collector is also able to calculate averages intended for plotting time series using a tool such as *gnuplot*, rather than be applied as a node label. Adding support for these averages as node attributes is not a priority at this point, but will be implemented in the near future. We describe the supported calculations in Section 5.1, where we further describe the use case intended for this feature of VIZPUB.

4.1.3 Visualization Unit

The *Visualization Unit* is a machine running the Gephi Open Graph Viz tool [4]. This tool is able to interpret the .gexf file generated by the Collector and visualize the execution of the pub/sub system in question. Gephi provides a rich GUI-experience where the user may interact with the graph representation, apply layout algorithms, filter the graph, execute metrics, apply color and size based on graph properties and animate the graph evolving over time. The Gephi software architecture is highly modular

and supports extensions via plugins, some of which are available in a official plugin marketplace.[2]. New metrics, filters or added functionality such as database support may be implemented through plugins by developers, and published to the marketplace free of charge.

Gephi provides many tools and components which are useful when inspecting and analyzing pub/sub overlays visually. These are what we consider the most important components:

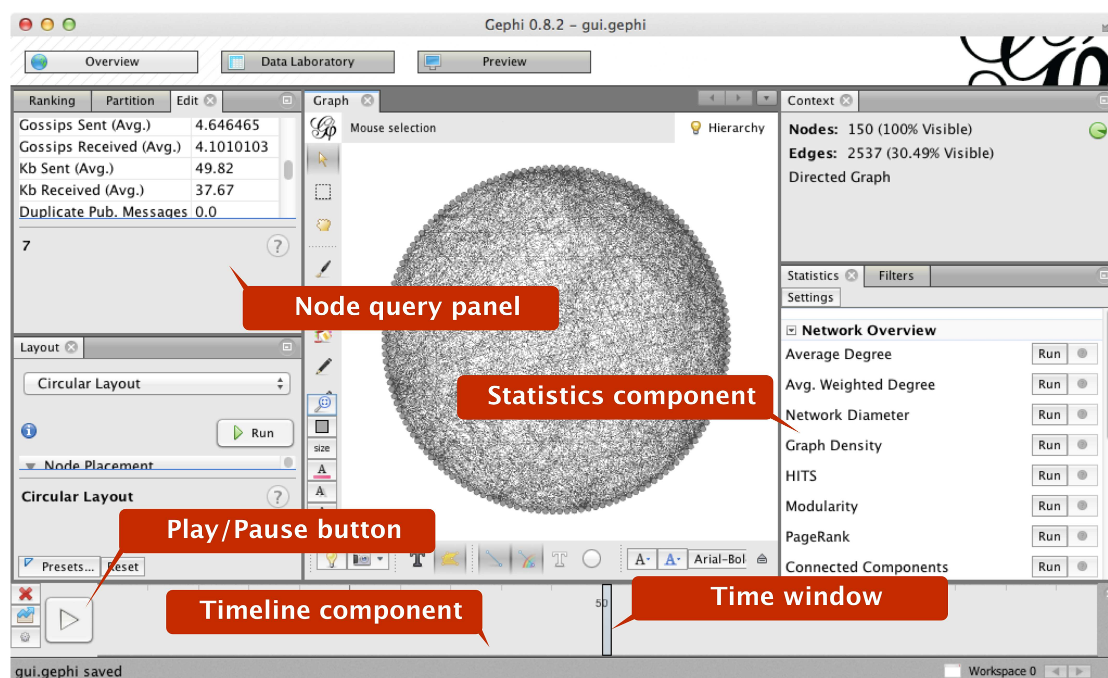


Figure 4.3: Snapshot of Gephi GUI with several important components annotated

Node Pencil and Edge Pencil Tools

These two tools enable the user to create nodes and edges by clicking in the graph view, which is the window where the nodes and edges are rendered. Edges can be directed or undirected, where direction is indicated with an arrow. These two tools combined enables building a graph by hand.

Building such graphs can be useful in order to reason, analyze or learn network algorithms such as event dissemination algorithms. For example, the user can start with a single node which can act as the event source, and build the topology with directed edges which depict the path of the message. This is very helpful in fully understanding such algorithms and the trade-offs they make. The user can

[2]http://marketplace.gephi.org

also add attributes to the nodes and edges either through the *Node Query Tool* or in the *Data Laboratory* component which also aids in visualising and understanding properties, as well as any drawbacks and advantages of such algorithms.

Node Query Tool

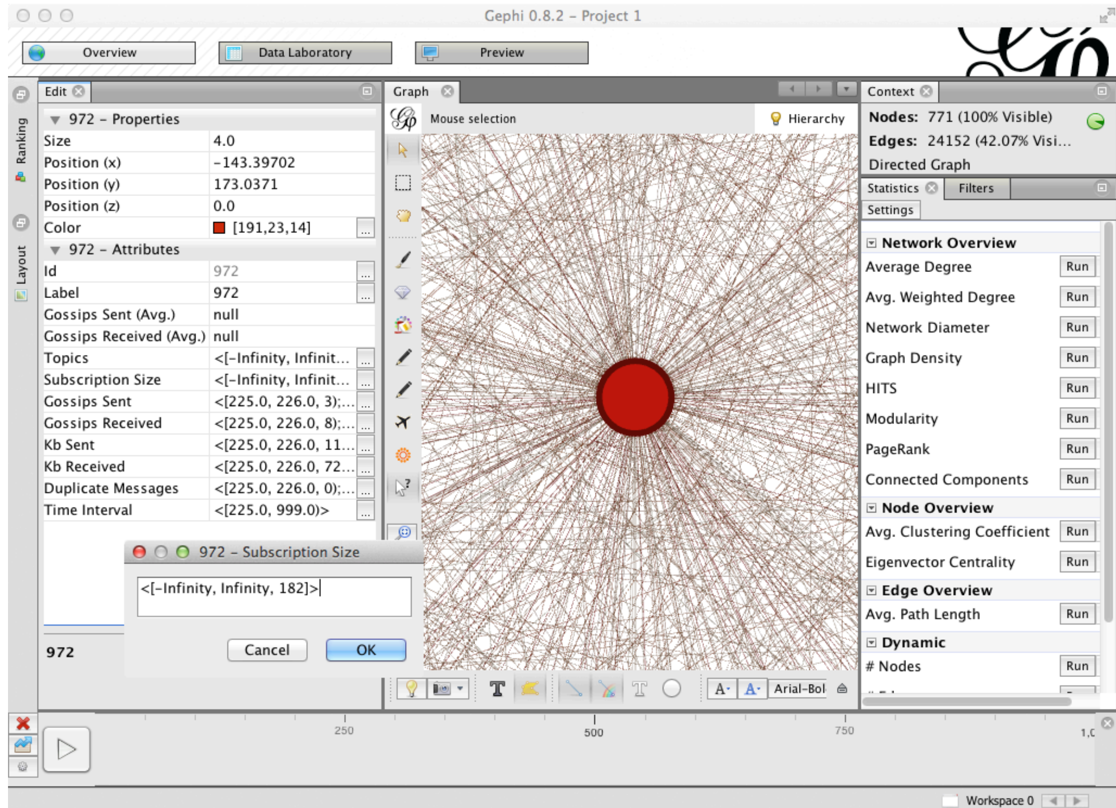


Figure 4.4: The Node Query Tool opens a panel (to the left) that can be used to inspect the attributes of a node, here it is used to inspect the subscription size of a high-degree node

With the Node Query Tool the user is able to click on a node in the graph view, and a panel will appear to the left with information regarding the *properties* and *attributes* of this node. Properties are data describing the visual parameters of the node such as size, position and color, while attributes include data such as node id, default label, the time intervals in which the node exists as well as any additional user defined attributes. In our case, examples of such user defined attributes would include topics, subscription size and number of control messages sent and received. The Node Query Tool can be seen in Figure 4.4, where it is used to inspect the *subscription size* (i.e. number of topics the node subscribes to) of a high-degree node.

Both the properties and attributes of the node are editable through the Node Query Tool. The user may select a property to change the visual representation of the node, or select an attribute in order to change its value. The *Time Interval* attribute in particular is interesting to edit, as it represents the points in time in which a node exists in the graph view. For example, it could be interesting to edit the Time Interval attribute for certain nodes in order to see how it affects a particular metric as well as the structure of the overlay topology in general.

Shortest Path Tool

With the shortest path tool selected, the user may click on two nodes in the graph view, and if there is a shortest path between them, this path will be highlighted with a color. This is useful in order to reason about the relationship between key nodes in the graph, or to compare shortest path between several pairs of nodes.

Heat Map Tool

The *Heat Map Tool* enables the user to click on a node in the graph view and color its neighborhood based on the edge weight distance between them. More specifically, it sets the node color intensity lower for more distant nodes and stronger for nodes that are closer. Edge weight is a standard edge attribute that is by default set to 1. This means that in the default case, the visualization will represent the hop count distance from the particular node selected by the user. However, the edge weight can be edited by the user in order to represent other properties of a system. For example, imagine setting the edge weight to represent network latency between two nodes. In this case, a neighboring node which is adjacent to the selected node would have a lower color intensity if the latency between them is higher than another neighboring node which is further away in terms of hop count.

Timeline Component

The *Timeline Component* introduces an animation scheme for temporal graphs. The user may choose playback parameters such as *time interval size*, *step size* and *playback speed*. The Timeline will automatically filter out a subgraph defined by the upper and lower bound of the interval. The evolution of the dynamic graph will then be animated by moving these bounds by the distance defined by the step parameter. The delay between each step is decided by the playback speed.

The Timeline enables the user to visually inspect the change in graph topology over time, as well as visualize and inspect node and edge attributes of the graph through both color, size and text labels which is able to change dynamically as part of the graph model animation. The Timeline also enables jumping to a specific point in time and investigating the corresponding subgraph and its properties by changing the upper and lower bound of the Time Interval.

Statistics Component

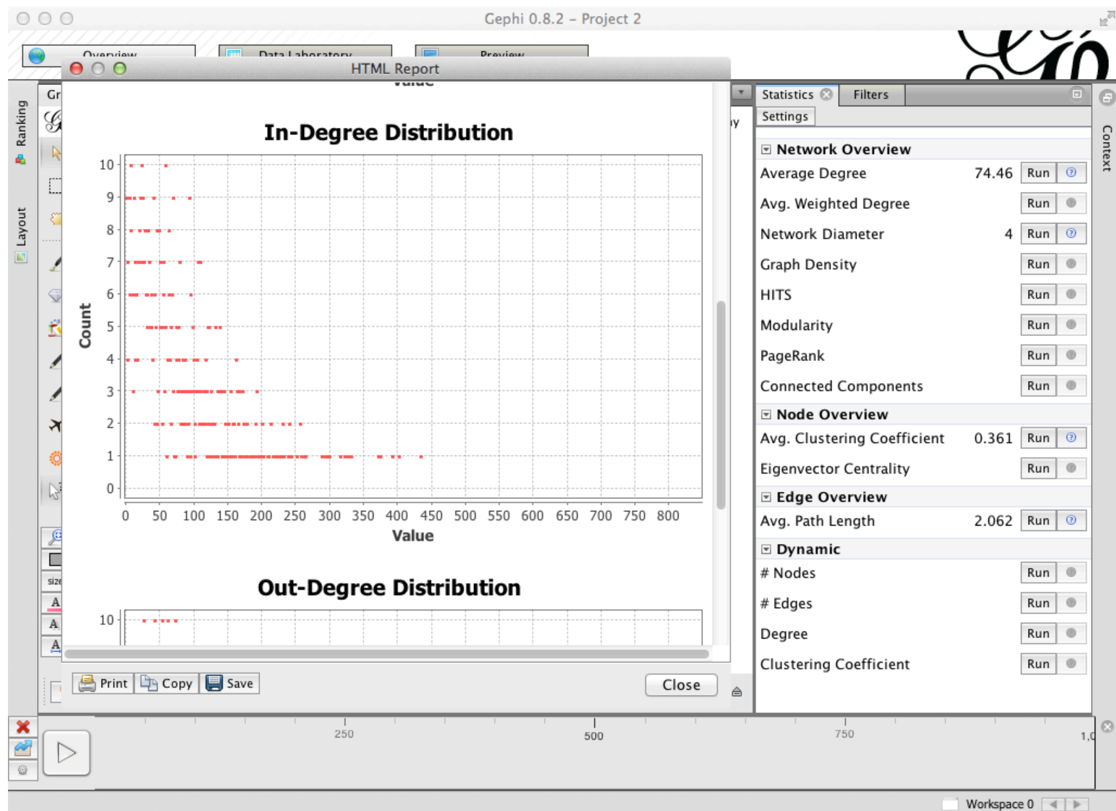


Figure 4.5: The Statistics component is able to produce a HTML report which includes plots describing the distribution of values across the nodes

The *Statistics Component* enables graph topology analysis by executing metrics on the graph. There are two types of metric algorithms in Gephi: *static* and *dynamic*, where the former calculates a single value based on the currently defined time window, while the latter calculates a time series of values. When executing a dynamic metric, the user must define the time window size and tick. These parameters have the same functionality as the step parameter when using the *Timeline Component*. When the metric executes, the time window will iterate through the entire time range of the simulation, calculating a value at each step. When finished, a time series is plotted and displayed for the user. Such reports are also generated and displayed when calculating static metrics. In the case of static metrics, scatter plots depicting the distribution of values are displayed, as seen in Figure 4.5

The Statistics component include several metrics which are relevant to pub/sub overlays. Useful static metrics include, but are not limited to:

- Degree (In/Out/Avg./Max/Min/Distr.)
- Avg. Cluster Coefficient
- Centrality (Beetweeness/Closeness/Eccentricity)
- Average Path length
- Radius
- Network Diameter
- Number of Shortest Paths

Out of these, only degree and the clustering coefficient metrics have dynamic versions, where both calculates the average value over time. The average for dynamic metrics are calculated by dividing the sum of all node attribute values with the total number of nodes in both cases.

Ranking Component

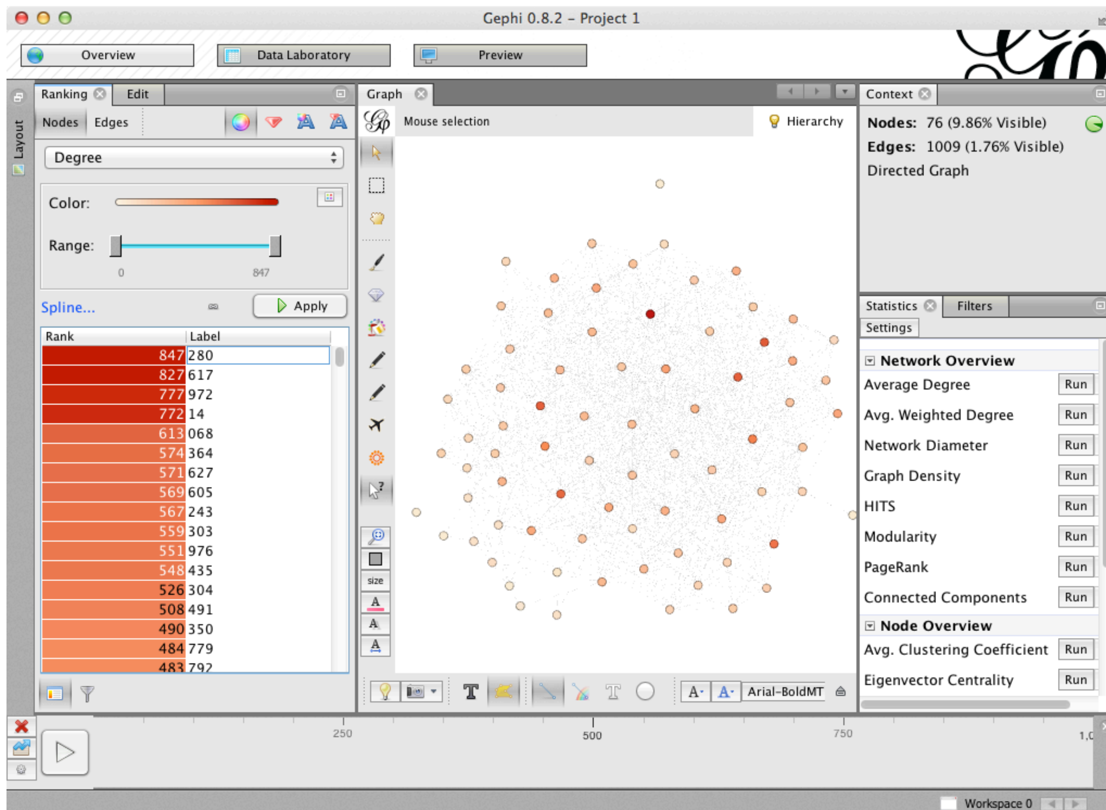


Figure 4.6: The *Ranking Table* (to the left) in the Ranking Component may be used to sort nodes by attribute value. In this screenshot it is used to rank nodes by degree in descending order.

The *Ranking Component* is a key feature of Gephi which enables visualization based on node or edge attributes in form of color and size. When coloring nodes or edges, the ranking component will apply a gradient over the range of attribute values. The ranking component also include a ranking table, as seen in Figure 4.7, where the user may sort nodes based on the specified attribute value. This is useful for quickly finding the nodes with maximum and minimum value, which is a quick way of identifying bottlenecks in the system or potential load balancing issues.

The Ranking Component also includes an auto apply feature, which supports visualising attributes dynamically while playing back the graph via the *Timeline Component*. More specifically, the node labels and color will update with every step of the animation.

Layout Component

The *Layout Component* enables the user to execute algorithms that calculates the position of the nodes. The user is able to adjust the parameters of these algorithms in order to manipulate the visual layout. The different algorithms emphasize different aspects of the topology. One example is the so-named *Force Atlas* layout algorithm which simulates the effect of gravity on the nodes, where linked nodes attract each other and non-linked nodes are pushed apart. This particular algorithm is useful for visually detecting clusters and communities. Another useful algorithm is the *Circular Layout* algorithm, where nodes are positioned in a circle ordered on a specific node attribute selectable by the user. This is useful in order to visualize node rankings on particular attributes. We use the circular layout extensively, as nodes in PolderCast are organized into a ring structure, where they are ordered by their unique id.

Filter Component

Filters may be applied to the graph in order to strip away nodes or edges on the basis of their attributes. Filters may strip away nodes based on a value range if the attribute type is a number, or they can be matched on a regular expression if the attribute is encoded as a string. Filters can also be combined through special operator filters representing set operations such as union and intersect.

Filters are an essential mechanism in order to analyze subgraphs. For example, in order to calculate *topic diameter* in pub/sub systems, a subgraph can be filtered out based on a *topic attribute*. Then, the diameter metric can be executed on this subgraph, which will result in the diameter of the particular topic.

Data Laboratory Component

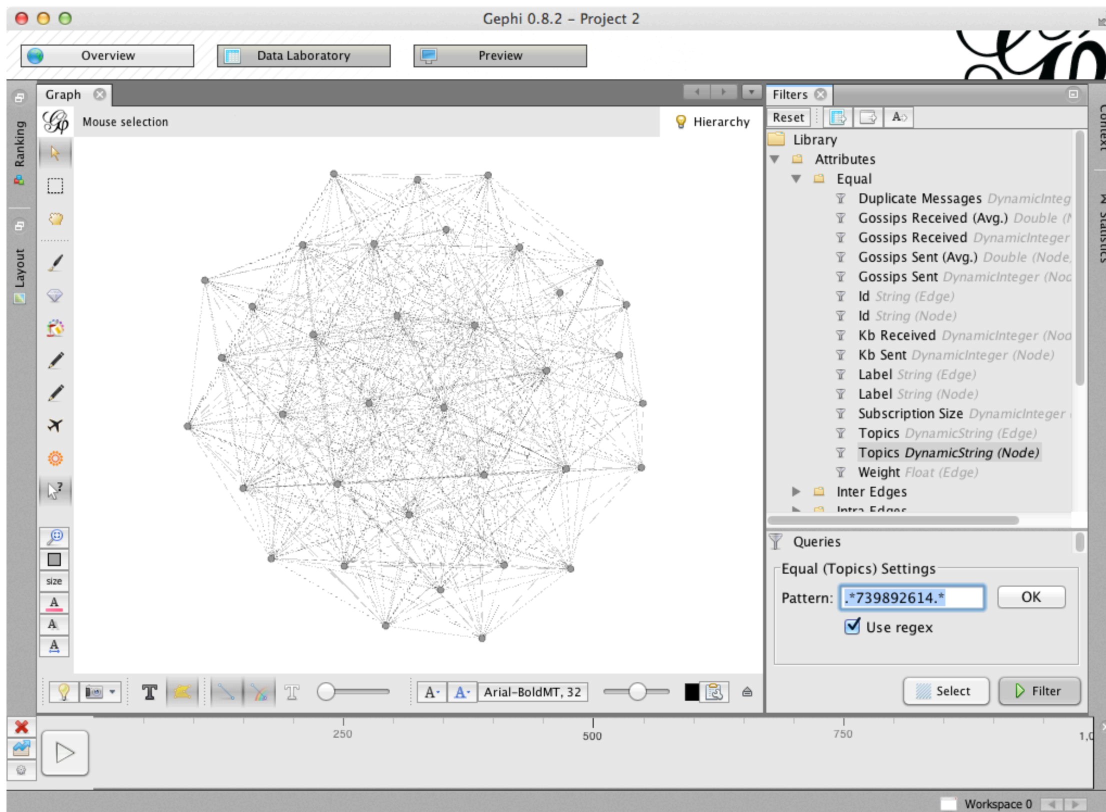


Figure 4.7: The Filter Component (to the right) enables users to strip away unwanted nodes and edges, in this screenshot it is used to display the sub-overlay for a specific topic by using a *regular expression*.

The *Data Laboratory Component* enables the user to work with the node and edge attributes of the graph. This component provides the user with separate table views of node and edge attributes, as seen in Figure 4.8. Each row in this table represent a node or edge, and each column an attribute. Columns may be added or removed by the user. The Data Laboratory also provides functionality for manipulating columns such as merging two columns or creating new columns based on the data from selected columns. Attribute data in columns that are static (i.e. has no lower or upper time interval bound associated with them) can be converted to dynamic through this component. Also, resizing or coloring all edges or nodes is possible through the laboratory by selecting all rows and right-clicking. In addition to this, the Data Laboratory also enables the user to export the data to file for further statistical analysis.

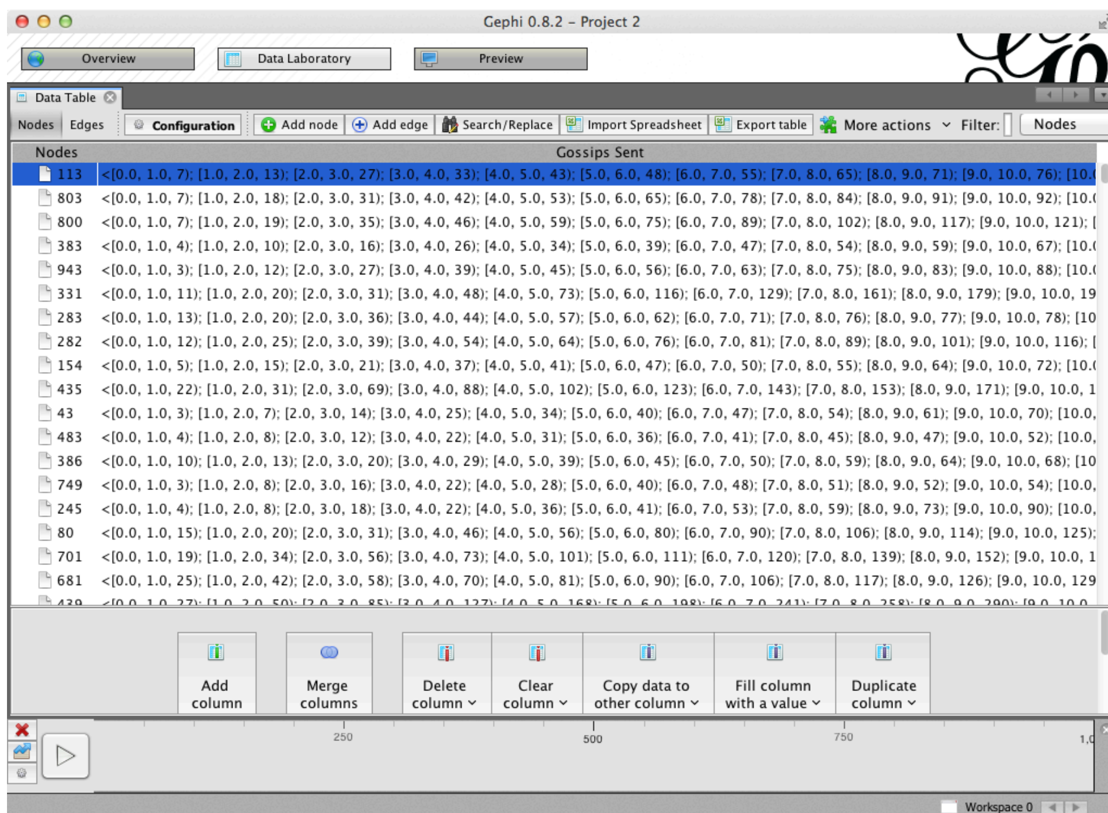


Figure 4.8: The Data Laboratory Component provides a table overview of metrics as well as the capability of exporting data.

4.2 Examples of Visualizations

In this section, we present a number of visualization produced by VIZPUB. The examples we provide in this section include both visualizations of overlay structure as well as hop-by-hop message dissemination. We implement a reporter interface both for PolderCast as well as Scribe and provide examples of visualizations for both protocols where we also compare them visually on various performance metrics. Both protocols are implemented using the PeerNet P2P simulator by updating existing PeerSim implementations of Scribe and PolderCast.

Many of these visualizations were part of our demonstration at DEBS 2014, and should provide some insight the benefits of using our tool, and what sort of possibilities there are in terms of visualizing overlays.

4.2.1 Data Traces Used in Simulations

We use publicly available datasets from Facebook [34] and Twitter [19]. The Facebook dataset consists of 3 million user profiles along with 28.3 million social relations

between these users. The Twitter dataset consists of 41.7 million distinct user profiles and 1.47 billion follow/followed relations. In both datasets users are modeled as topics. In the Facebook dataset, subscriptions are modeled after the friends list of the particular user. As relationships in Facebook are bidirectional, two topics will subscribe to each other. In Twitter, relationships are unidirectional, as users may choose to follow other users, but no user who is followed need to reciprocate. As a user is modeled as a topic, its list of followers are modeled as the subscribers of that topic.

Churn is based on the Skype super-peer network data trace [14], where 4000 nodes are tracked for joining and leaving timestamps for one month staring on September 12, 2005. Finally, we use the King dataset [15] in order to model latency between nodes.

4.2.2 Overlay Evolution During Churn

In Figure 4.9, 4.10 and 4.11 we visualize the overlay topology of 2000 PolderCast nodes during churn. This is one of the examples of visualizations we presented at DEBS 2014. Each figure describes the structure of the overlay at different reporter intervals. Figure 4.9 depicts the overlay at interval 0, where 132 nodes are up while the remaining 1868 are down due to churn. The snapshot in Figure 4.10 depicts the overlay at interval 250, after a considerable number of nodes have joined the network. More specifically, interval 250 consists of 1028 nodes, which means 896 nodes joined in the interim. It is also possible to observe the edges evolve over time. While the first interval consisted of 108 edges, interval 250 consists of 1028 edges. It is interesting to look at how the edges evolve, as they provide us with immediate visual feedback on properties such as clustering of nodes, graph density and node degree. Such properties can then be further analyzed using the Statistics Component in Gephi.

Visualizations of overlay structure also provide immediate information regarding the dataset being used. As mentioned, we encountered a scenario where an artefact in the dataset resulted in a disconnected component in the visualized overlay. Here we notice that many of the nodes are disconnected. This is due to a sampling bias, as the particular Facebook dataset sample used for this visualization contains 10,000 nodes while the simulation runs 2000 nodes. However, visualizations including a high number of nodes is not appropriate for print due to space restrictions, also visualizing a high number of nodes require more graphics rendering power than what we had available at the time. The visualization still serve as an example of revealing both properties of the overlay, as well as properties of the workload sample being used, and any potential biases included in it.

4.2.3 Visualizing Performance Metrics

VIZPUB supports visualizations of a number of performance metrics. These metrics are applied as attributes to both nodes and edges. The supported metrics are listed in Section 4.1.2. In Gephi, we can visualize these metrics by applying the corresponding attribute as a node label as well as use the *Ranking Component* in order to apply color to the nodes. Such visualization can be seen in Figure 4.12 which shows the average

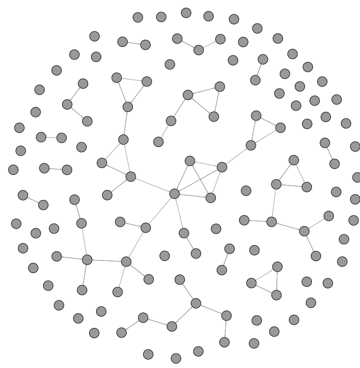


Figure 4.9: Overlay structure of PolderCast at interval 0

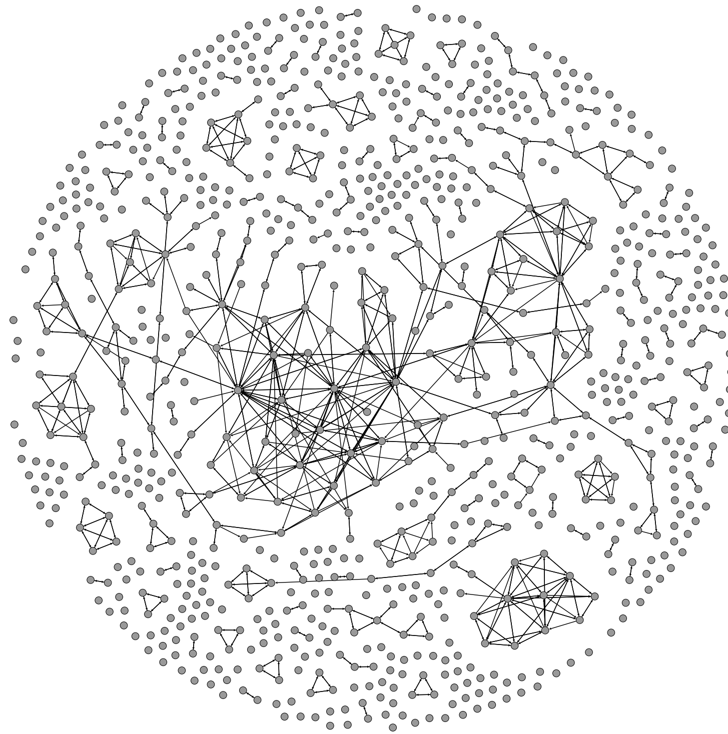


Figure 4.10: Overlay structure of PolderCast at interval 250

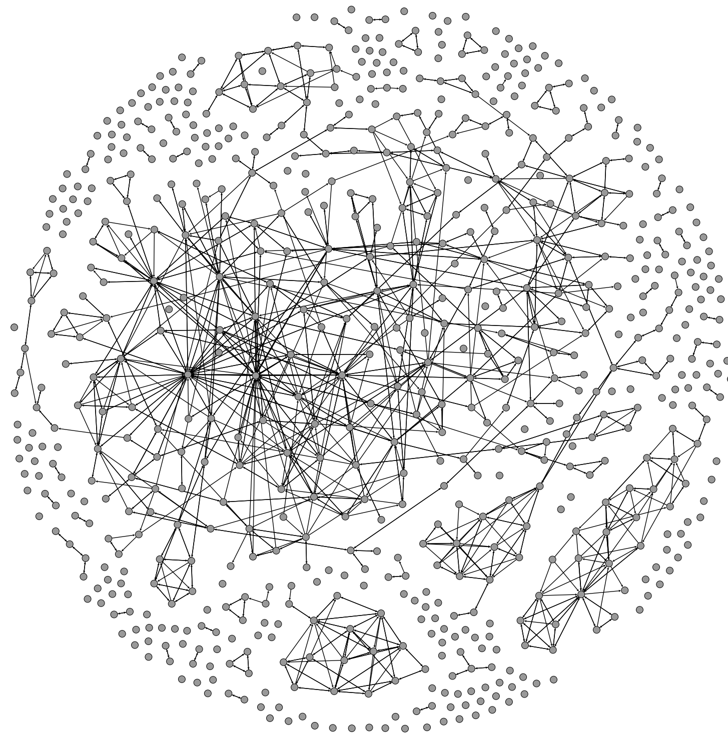
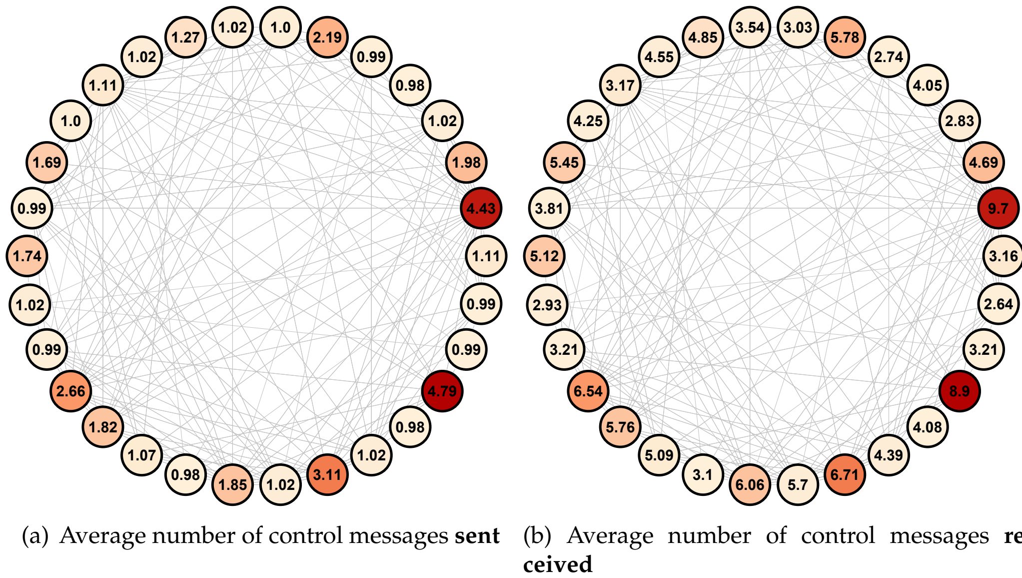


Figure 4.11: Overlay structure of PolderCast at interval 500



(a) Average number of control messages **sent**

(b) Average number of control messages **received**

Figure 4.12: Visualizations of average number of control messages sent and received per node in PolderCast

number of control messages being sent and received for each node in PolderCast. This visualization can be produced using the same .gexf file as the one which is used for creating the visualization of overlay structure during churn seen in Section 4.2.2.

Applying colors helps in discovering potential bottlenecks by highlighting overloaded nodes. Both the node labels and node colors will update during playback of system execution. As the control message metrics are running totals, the user will observe the nodes getting a deeper color towards the end of the playback. Users can choose the pause the animation any time they like, as well as jump to any point in time. The user is then able to inspect the overlay, run metrics using the *Statistics Component* as well as produce plots as seen in Section 4.2.5. The user may also export data to a .csv file using the *Data Laboratory* for further statistical analysis.

4.2.4 Publication Message Dissemination

In subsection 4.2.2 we illustrate one of the two types of visualizations it is possible to produce by leveraging VIZPUB, namely visualizations of overlay structure. Here we will describe the second: visualization of publication message dissemination.

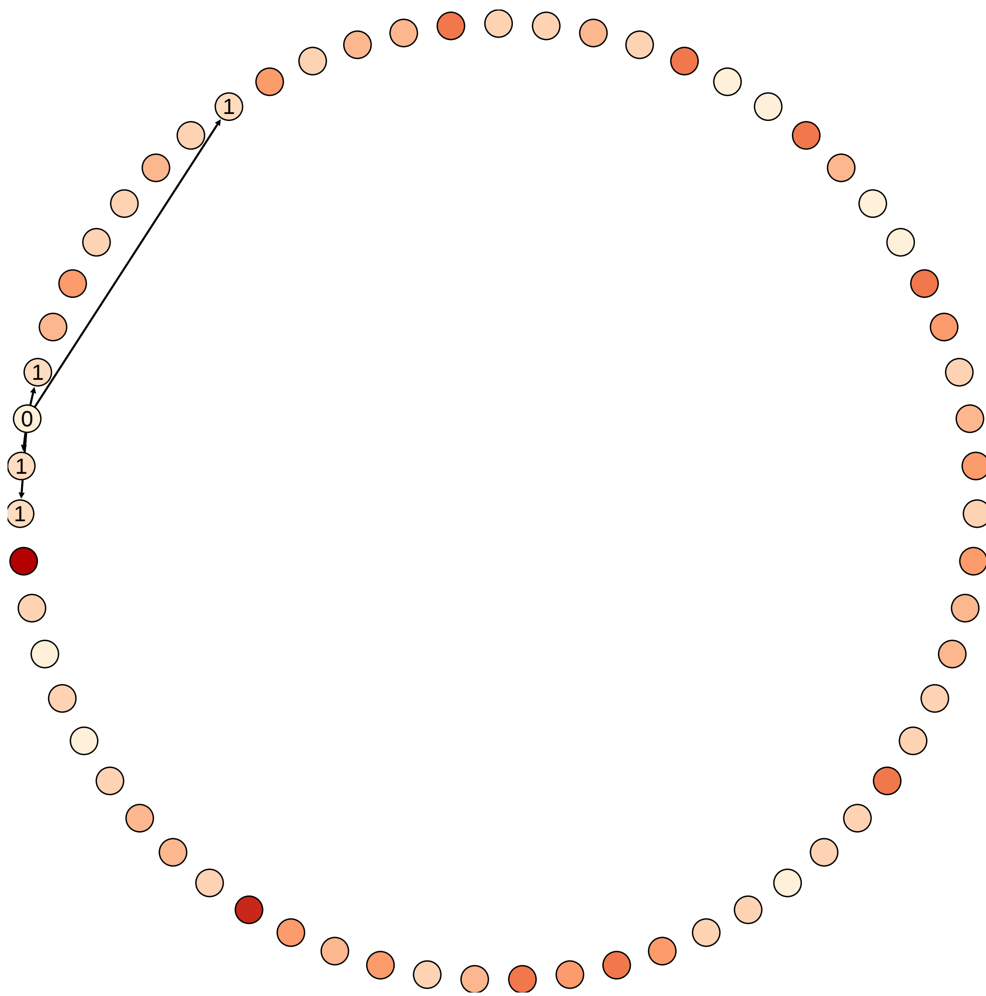
We run a PeerNet simulation in distributed mode with 100 nodes running the PolderCast protocol with a fanout set to $f = 2$. In order to scale the experiment, the simulation includes 10 machines, where each machine is running 10 PeerNet nodes. We publish a single message on the most popular topic which includes 63 subscribers. By analyzing copies of the publication message received on each node, the Collector is able to create a step-by-step visualization where directed edges are drawn as the message

is disseminated through the overlay. These edges depict the path of the message being disseminated. The visualization includes 189 edges in total, which indicates how many publication messages was sent by all nodes during the simulation.

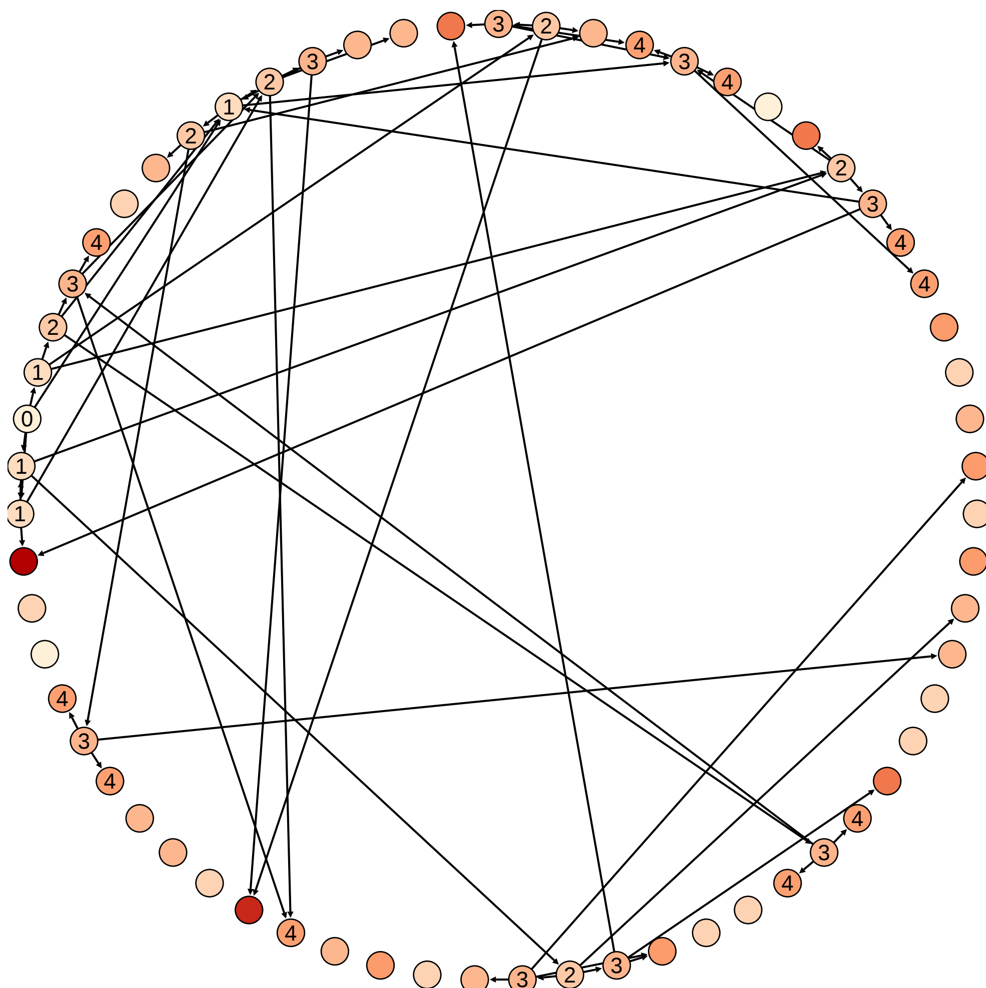
Figure 4.13 is a visualization of the dissemination in PolderCast after 1 and 4 hops, while 4.14 are snapshots of the visualization after the dissemination is finished. The nodes have been arranged in a ring using the *Layout Component* in Gephi. Also they are ordered by node id, making it easier to see whether an edge represent sending a message through a ring link or a random link. The labels indicate the hop number of the message received by that node. On the left side of the ring in Figure 4.13 and Figure 4.14 there is a node with the label "0", which indicates that this is the publishing node. A node with the label "1", means the node received a message at the first hop, a node with a label "2" that it received a message at the second hop, and so on. In addition to numeric labels, the nodes are color coded on a gradient. The deeper the color of the node, the higher the hop count, which also means it is further away from the publishing node. In Figure 4.14 the node furthest away is easily spotted on the left by its deep red color. This is the last node to receive the publication.

It might seem strange that the last node to receive the message is an immediate ring neighbor of a node who received the message on the first hop. Observe that in Figure 4.13 this neighboring node who received a message on the first hop does indeed send a message to the node in question. However, in Figure 4.14, which depict the dissemination after the message has been sent, no numeric label is applied to the node. This could give the impression that this node did not receive any message. However, this is not the case. The explanation behind this is latency. The latency between these two nodes is so high that by the time the recipient node received the message from its ring neighbor, the message was already received from another node. We choose to visualize this by drawing an edge to the node and then refrain from applying a numeric label. When we reach the point in the animation where this node receives a message for the first time, we apply the hop count of this message as a numeric node label.

When inspecting the dissemination algorithm of protocols visually, implementation details which could otherwise be overlooked become easily detectable. For example, in Figure 4.13 it can be observed that the publisher node sends a message to four nodes, even though the fanout is set to $f = 2$. This fanout indicate that the node should send the message to three neighbors, based on the description of the dissemination algorithm of PolderCast, summarized in Chapter 2. However, there is an implementation detail in PolderCast that is not mentioned in the original paper [30]. In the implementation of the PolderCast protocol, publishers send messages to one additional random node in order to boost the initial phase of the dissemination. Learning such implementation details is useful to both researchers and developers, and it is especially useful for students. We believe VIZPUB can be very valuable as an educational tool, as its grant students with the capability of controlling the dissemination by using the Timeline Component in Gephi. Students may pause the visualization at any point in time or jump to any step of the visualization in order to fully understand the benefits and drawbacks of the particular dissemination scheme being studied.



(a) After 1 hop



(b) After 4 hops

Figure 4.13: Visualization of publication message dissemination in PolderCast.

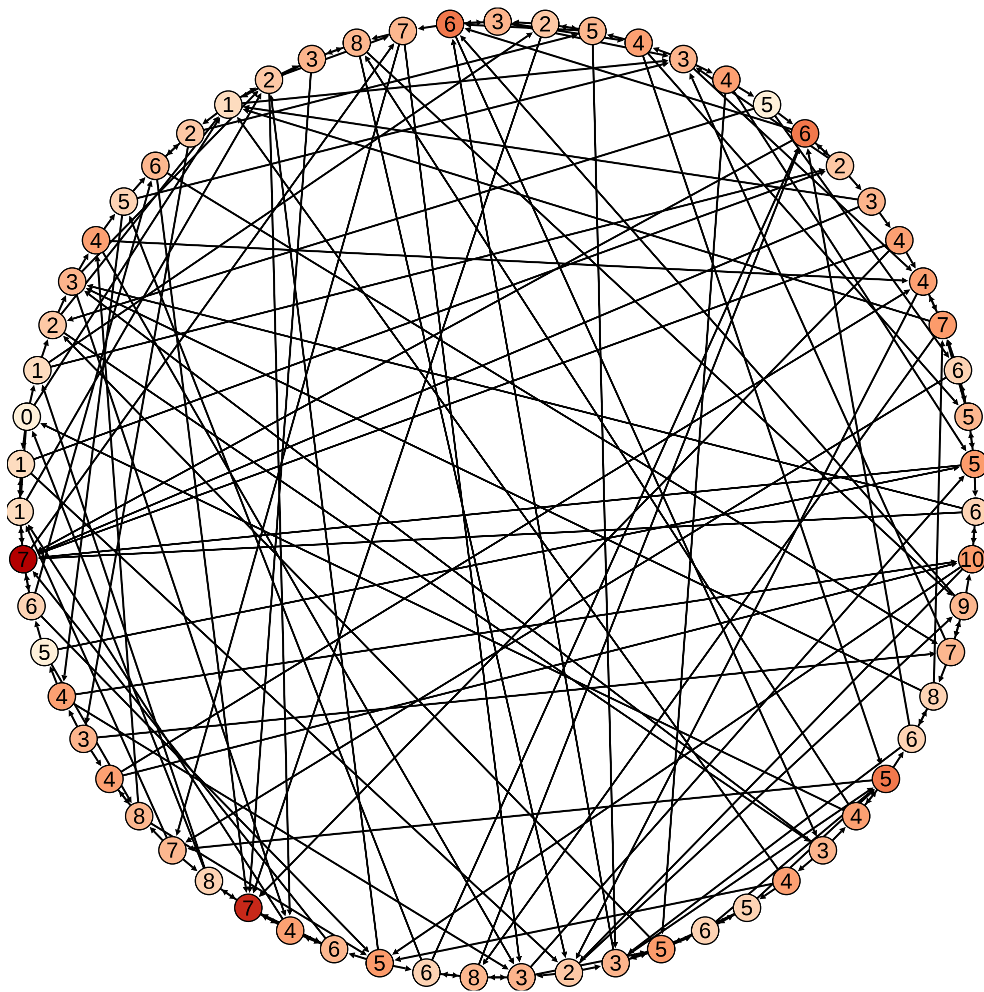


Figure 4.14: Visualization of PolderCast after the end of dissemination

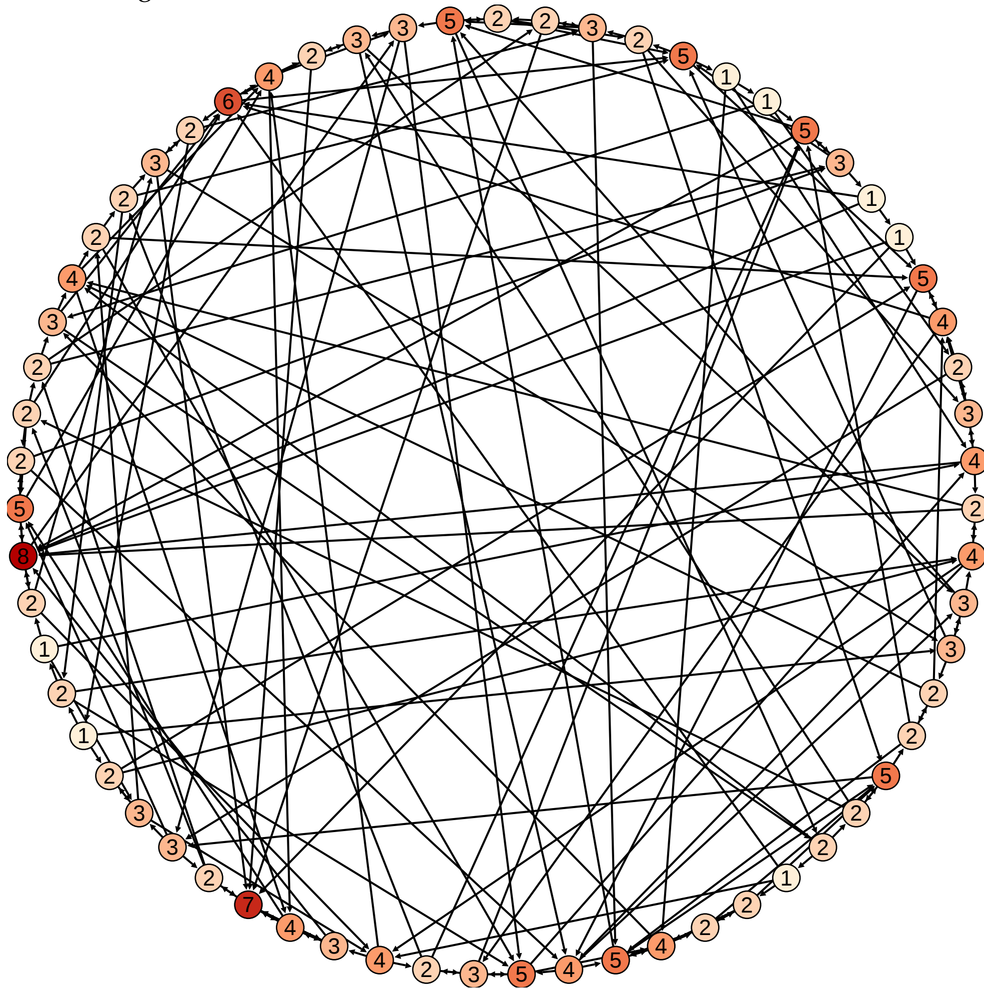


Figure 4.15: Visualization of duplicate publication messages received by each node in PolderCast

It is also useful to create such visualizations in order to discover issues or bugs with the dissemination algorithm. As observed in Figure 4.13, the publisher disseminates to four nodes, where two of these should be random neighbors. However, the publisher seemingly sends the message to three of its closest ring neighbors. This means a neighbor close to it have been chosen as a random neighbor. This might indicate a bug in the CYCLON module of PolderCast, which is responsible of providing the RINGS layer with uniform random neighbors. However, the dissemination happens at an early point of the experiment, more specifically, after 50 PeerNet cycles. Due to experimental settings, this might not have been enough time for the different layers of overlays in PolderCast to converge. Also, it could be a special case, which resulted in the publisher being a bit "unlucky" when picking a random neighbor in this particular scenario. Regardless, visualizing dissemination leads to these types of interesting observations, which might lead to even more interesting findings with regards to system behaviour. For example, another interesting observation is how a fanout of $f = 2$ leads to a special case when disseminating messages in PolderCast. More specifically, as mentioned in Chapter 2, a node running PolderCast will forward a message to both ring neighbors and $f - 2$ random neighbors if it received a message through a random link. But if $f = 2$, then the number of random neighbors would be zero, since $f - 2 = 0$. However, PolderCast will always include a minimum of one random link when it forwards a message. This is yet another implementation detail which could be hard to catch without being able to inspect the dissemination protocol visually.

One of the trade-offs we mention in Chapter 3 is the one between the number of duplicate messages received and the robustness of message delivery. In Figure 4.14 we can quickly confirm visually that some of the nodes have a high degree. This indicates that the number of duplicate messages this these nodes received is high. A certain number of duplicate messages in epidemic dissemination is to be expected, but a balance should be struck between number of duplicate messages and reliable delivery. If there are too many unnecessary messages being sent, scalability in terms of bandwidth suffers.

Deriving the exact number of duplicate messages received by each node in the visualization of publication message dissemination is simple. As each directed edge represent a message being sent, all we need to do is calculate the in-degree of each node using the Statistics Component in Gephi. The result can be seen in Figure 4.14. This visualization indicate that PolderCast does indeed introduce a rather high number of duplicate messages being received on each node. However, it is only an indication and nothing more as this visualization traces a single publication message on a single topic. We should not use this observation to make any sweeping conclusions regarding the general case. However, such an indication can be useful in order to guide researchers and developers towards potential issues or bugs. We believe VIZPUB is a useful tool in this aspect.

4.2.5 Comparing Pub/Sub Systems Visually

In Figure 4.16(b) and Figure 4.16(a) we visualize the structure of PolderCast and Scribe. We run PeerNet experiments in simulation mode, consisting of 1000 nodes, 1000 reporter intervals and 1000 PeerNet cycles. The visualizations represent the structure of the overlays at reporting interval 1000. Subscriptions are modeled after the Facebook dataset found in [34]. The resulting .gexf files are imported into Gephi in order to analyze the overlay structure both visually as well by calculating certain metrics using the Statistics Component. The calculated values are described in Table 4.3. *Number of Nodes* and *Number of Edges* are the total number of unique nodes and edges that exists during the experiment, while the other values are calculated averages across the duration of the experiment.

	PolderCast	Scribe
Total Number of Nodes	833	833
Total Number of Edges	18629	883
Average Degree	22.36	1.06
Average Graph Diameter	10	22
Average Path Length	2.96	8.9
Average Graph Density	0.027	0.001
Weakly Connected Components	63	280
Strongly Connected Components	83	623
Average Clustering Coefficient	0.369	0.003

Table 4.3: Calculated metric values for PolderCast and Scribe, calculated using the *Statistics Component* in Gephi.

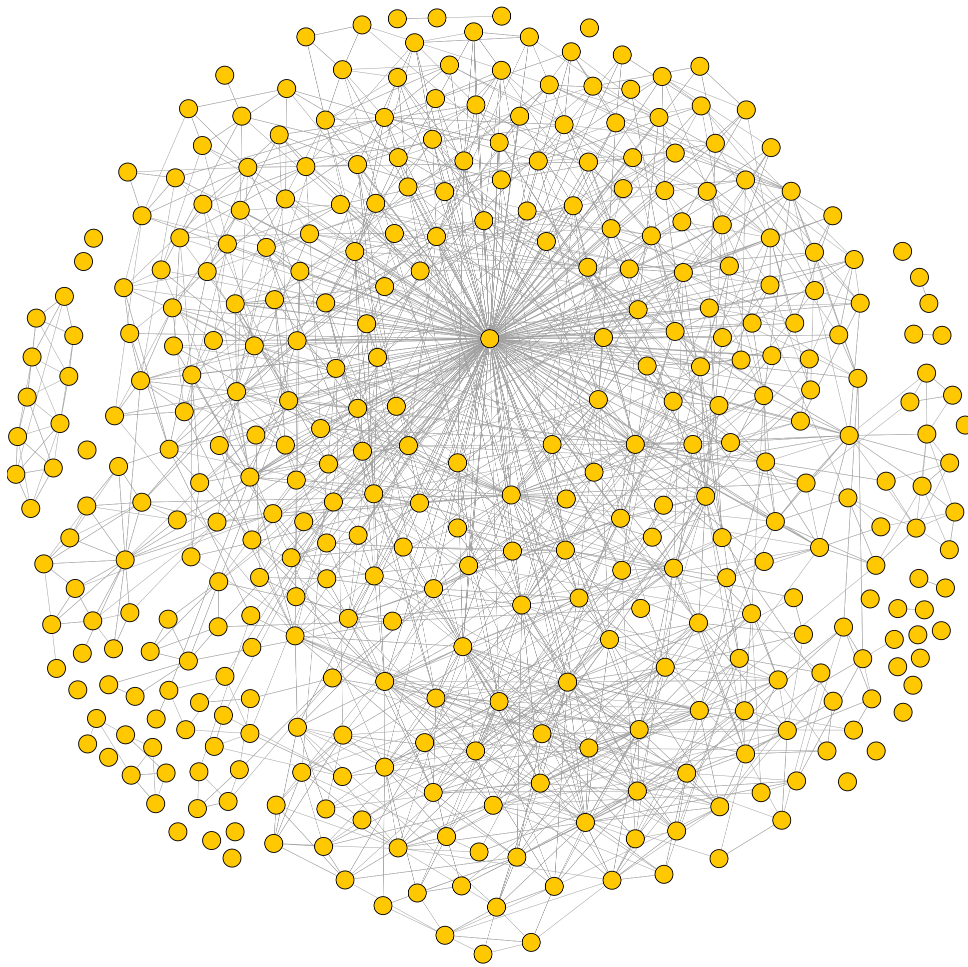
Inspecting the overlay structure visually enables us to derive certain information which can guide us in further investigations. For example, by visually comparing the two overlays in Figure 4.16, we can observe that the overlay of PolderCast seem to include a lot more edges, as well as being more dense than the overlay in Scribe. Both these visualization consists of 362 nodes, but the PolderCast overlay seen in Figure 4.16(b) consists of 2307 edges, while the Scribe overlay, seen in Figure 4.16(a) consists of 193 edges. The overlay in PolderCast has almost twenty times the number of edges. This is consistent with the total number of edges across the duration of the experiment seen in Table 4.3. The higher number of edges should also mean PolderCast also has a much higher average degree as well as a higher graph density. Executing these metrics using the Statistics Component reveal that PolderCast indeed have a much higher degree, again almost twenty times that of Scribe. The *graph density* is also much higher in PolderCast. Graph density is measured as how far away the graph is to being complete, where 1 is a complete graph. The lower density of Scribe is a natural consequence of the lower number of edges. The difference in edge count could be due to the average node degree having a lower upper bound in Scribe, as described in Chapter 3. More specifically, the average node degree in Scribe is bound by the

logarithm of the number of nodes in the overlay $O(log|\mathcal{V}|)$, while in PolderCast it is in the order of the number of subscriptions $O(|\mathcal{T}_v|)$. However, the difference in number of edges could also be indicative of PolderCast being more resilient to churn, as the dissemination tree structures in Scribe are brittle and need constant maintenance.

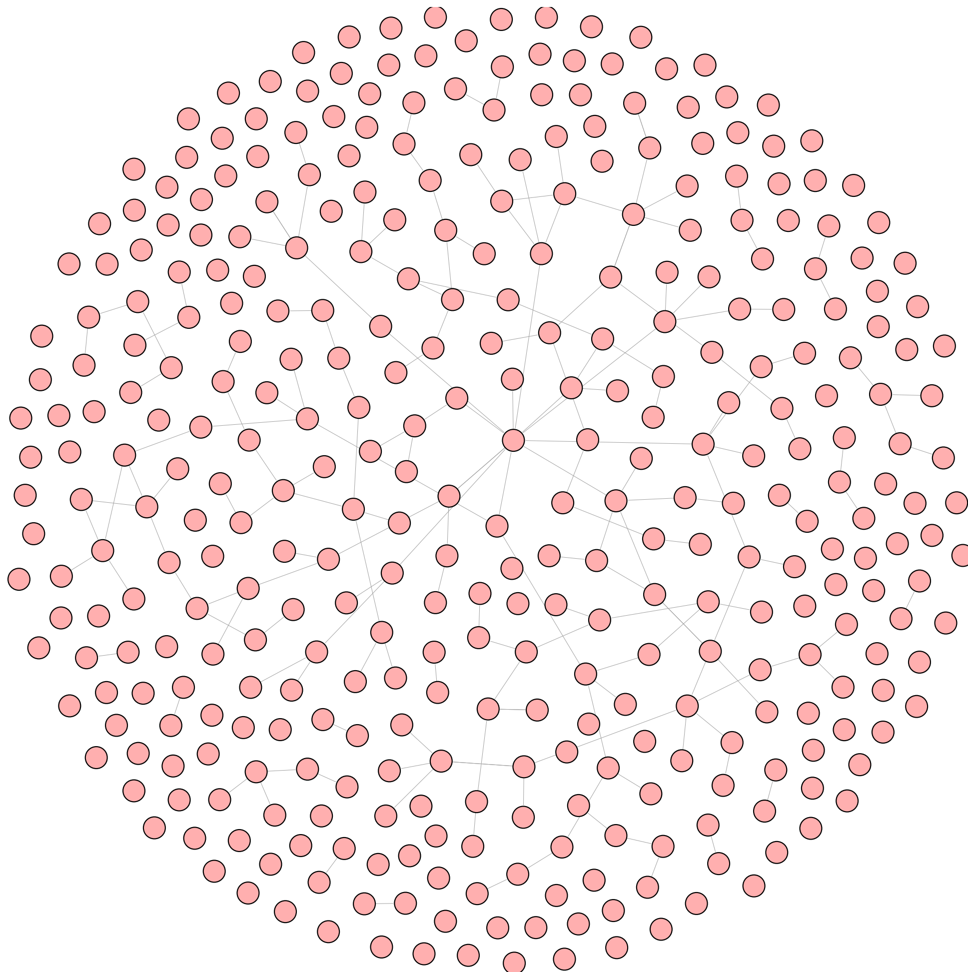
The fact that the number of edges in PolderCast is much higher, also means the overlay is better connected. This is evident if we look at the number of connected components. Strongly connected components have a directed path between every pair of nodes, while weakly connected components are connected through undirected edges. As seen in Table 4.3, the Scribe overlay is more partitioned, as it includes a higher number of graph components, both strongly connected as well as weakly connected. The higher number of graph components in Scribe is likely due to the topics being organized into separate dissemination trees, where the paths between the rendezvous nodes are connected at the DHT layer. Also, disconnected nodes are caused by subscribers not having any topic neighbors online due to churn. In fact, the churn settings of the experiment might have had an detrimental effect on Scribe, as it relies on tree structures which are more brittle and in need of constant repair when facing a high degree of churn. But this illustrates one of the difficulties designers of such pub/sub systems face: how do you balance robustness of the overlay while maintaining a low node degree? The two visualizations in Figure 4.16 should serve as examples of two different approaches to this question.

Note from Table 4.3 that both the average diameter as well as overlay path length is shorter in PolderCast than in Scribe. This is a natural consequence of the number of edges. It indicates a better connectivity in PolderCast, which is beneficial in terms of message dissemination.

Looking at the overlay structures in Figure 4.16(b) and Figure 4.16(a) also indicate that some nodes are overloaded. The PolderCast overlay in particular seems to have a very overloaded node in terms of degree, visible in the upper middle of Figure 4.16(b). Using the capabilities of VIZPUB, we are able to look further into the issue of load balancing. Figure 4.17(a) and Figure 4.17(b) visualize the degree of the nodes in the overlay by color. The deeper red the color of the node is, the higher its degree. This is useful for spotting potential bottlenecks in the system, as well as getting a general overview of the degree distribution. We can immediately observe that Scribe seems to have a more even distribution of color, which implies a better distribution of degree. But in order to determine the specific values, we need more than a colored graph. One option is to apply the degree value to each node as a text label. However, we skip this option as it is not suitable for print. But in a scenario where the researcher explores the overlay on a computer screen, this is a useful way of quickly determining attribute values of nodes. Luckily, the Statistics Component in Gephi includes the option of displaying scatter plots of degree distribution, as well as storing these plots to the hard drive as image files. Figure 4.19(a) and Figure 4.19(b) describes the undirected degree distribution of PolderCast and Scribe. Note that these values are averages across the duration of the experiment, while the visualizations found in Figure 4.16(b) and Figure 4.16(a) only represents the structure of the overlay at interval 1000. Looking

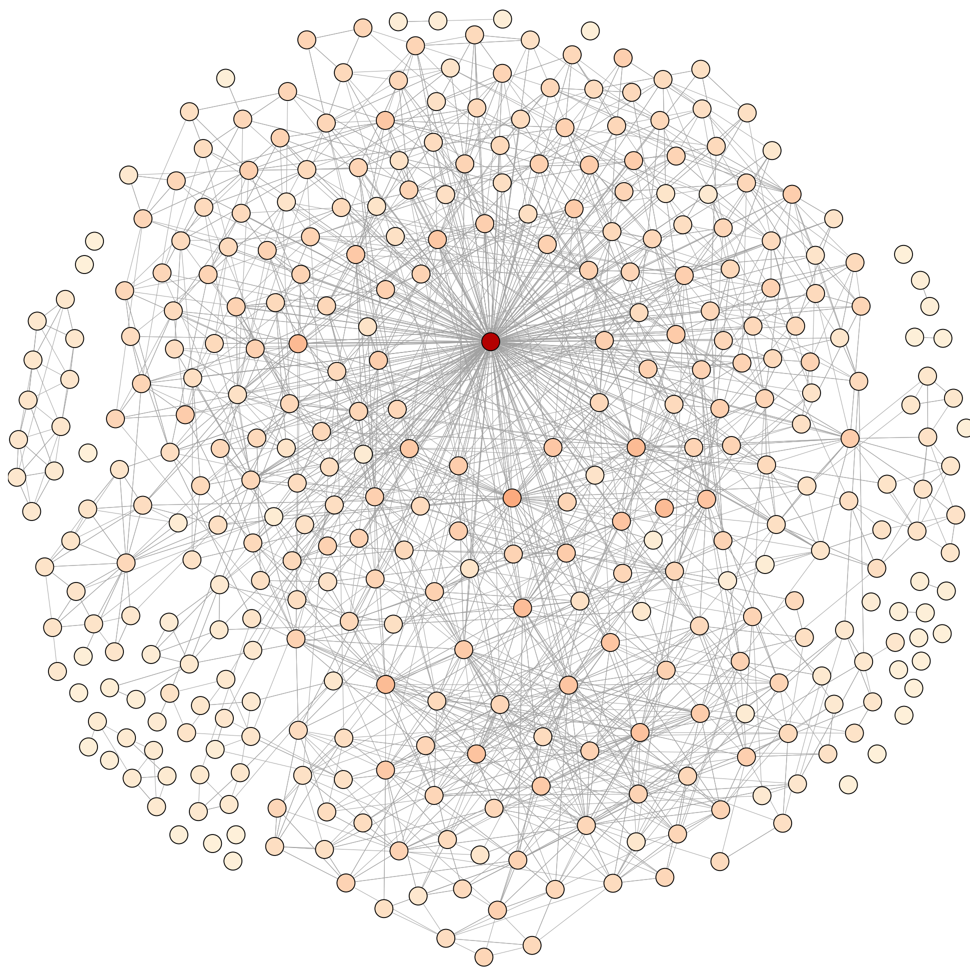


(a) PolderCast

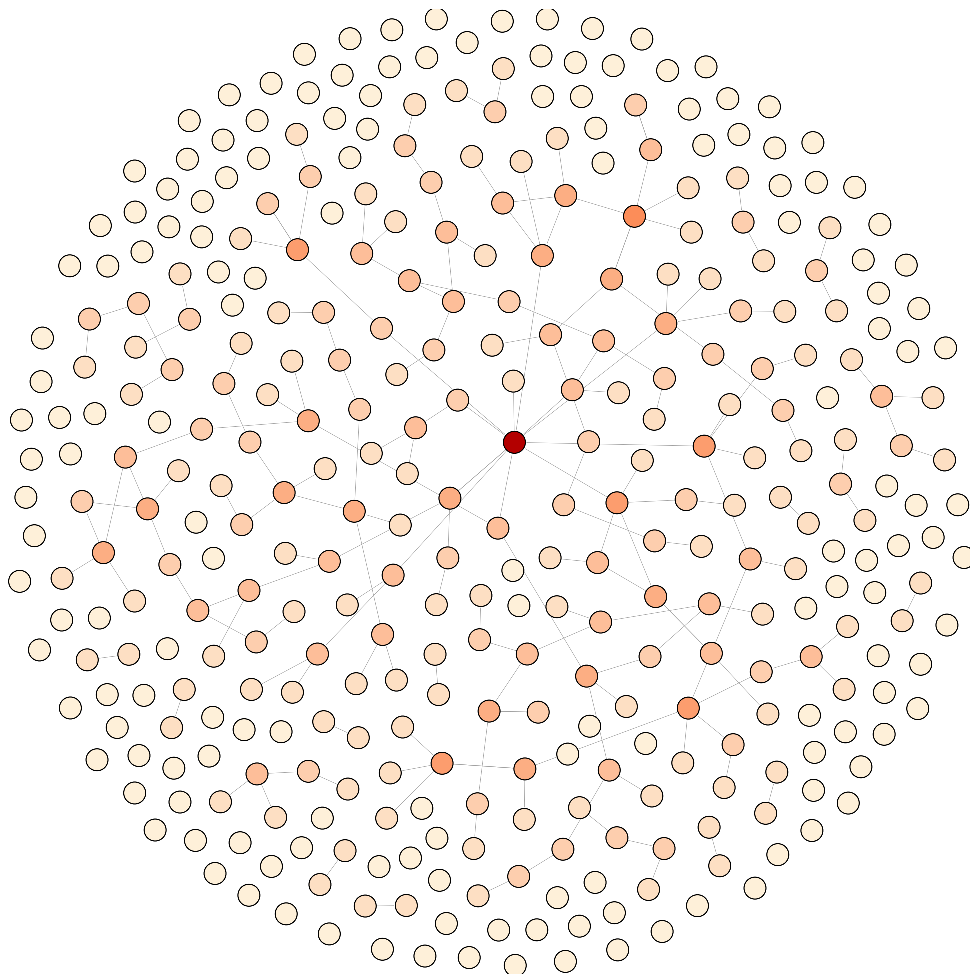


(b) Scribe

Figure 4.16: Visualizaton of the **overlay strucure** of PolderCast and Scribe after 1000 intervals



(a) PolderCast



(b) Scribe

Figure 4.17: Visualization of **degree** in PolderCast and Scribe after 1000 intervals

at the distributions in Figure 4.19(a) and 4.19(a),we can see that both are skewed to the right. However, the range of values are more tightly bound in Scribe, indicating a fixed degree. As mentioned, PolderCast has a average node degree proportional to the number of subscriptions, which results in a larger range of values than what is seen in Scribe. In fact, by using the *Rank Table* in Gephi, we can quickly find the maximum node degree in the overlay at interval 1000, which can serve as an indication of the difference in node degree. Doing this, we find that in PolderCast the maximum node degree at interval 1000 is 648, while in Scribe it is 193.

The high-degree node found in the PolderCast overlay in Figure 4.16(b) might indicate that this particular node has a much higher number of subscriptions then the other nodes in the overlay. We can confirm this suspicion by using the *Node Query Tool*, which shows us that this particular node subscribes to 466 topics, which is is 418 topics more than the node which is ranked second in terms of subscription size. This is consistent with the way degree grows with the subscription size in PolderCast. Scribe might have an advantage in terms of node degree if there are many nodes in the system with a high number of subscriptions. The same node in Scribe, seen at the middle of the overlay in Figure 4.16(a), has node degree of 12.

Figure 4.19(c), 4.19(e), 4.19(d) and Figure 4.19(f) describes the distribution of average directed degree across the duration of the experiment per node in PolderCast and Scribe. Both in PolderCast and Scribe the distribution of in-degree and out-degree seems to be quite balanced and both distributions seem to have a long tail. However, there seems to be an outlier in the PolderCast distributions, which would represent a node with a very high average node degree. This is most likely the high-degree node mentioned earlier. If we look back at the undirected distribution found in Figure 4.19(a), we can barely make out this outlier in the far right of the plot, although it is difficult to spot. Unfortunately, Gephi does not provide any options for configuring these plots, otherwise we could be able to increase the range of the x values in order to see this data point properly.

In addition to inspecting degree distribution, Gephi also provides us with the possibility of plotting time series. These are produced after executing the *dynamic* metrics in the Statistics Component. The time series can tell us additional information regarding the evolution of the overlay structure. We can easily produce a chart plotting the network size, i.e. the number of nodes in the overlay, as seen in Figure 4.2.5. Here, a spike in the number of nodes joining the network can be seen around interval 100. The effect of this spike is easily observed in most of the plots in Figure 4.20. Naturally, the number of edges also increase sharply at this point in time, as seen in Figure 4.20(a) and Figure 4.20(b) as well as degree seen in Figure 4.20(c) and Figure 4.20(d). Note that the average degree in Scribe drops at around interval 350, where many nodes leave the network. This is not the case in PolderCast, where the average node degree fluctuates less than in Scribe. This yet again indicates that Scribe is more vulnerable to churn. However, in both overlays the number of edges drop around interval 400, where a lot of nodes are down due to churn. Still the node degree in PolderCast remains stable. This might serve to illustrate the robustness of the PolderCast overlay.

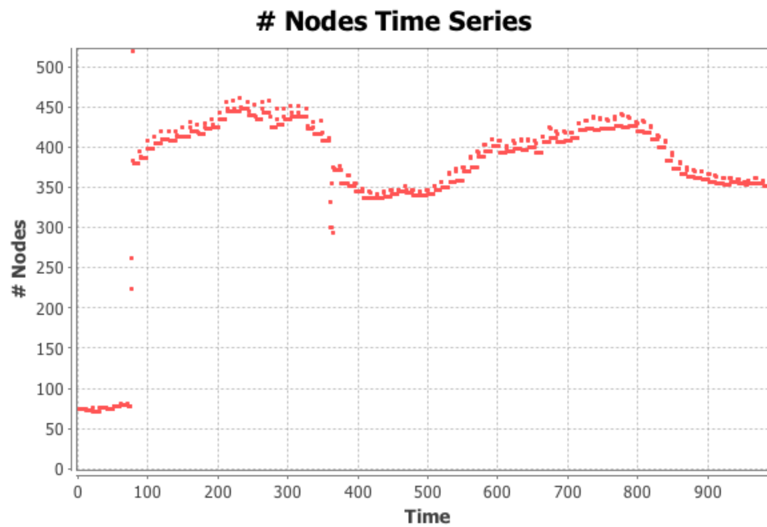
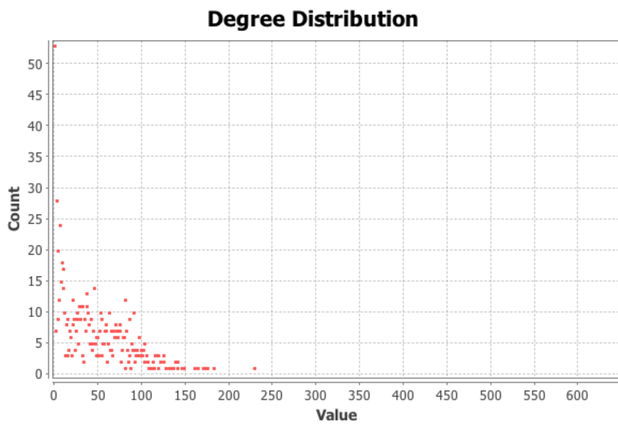


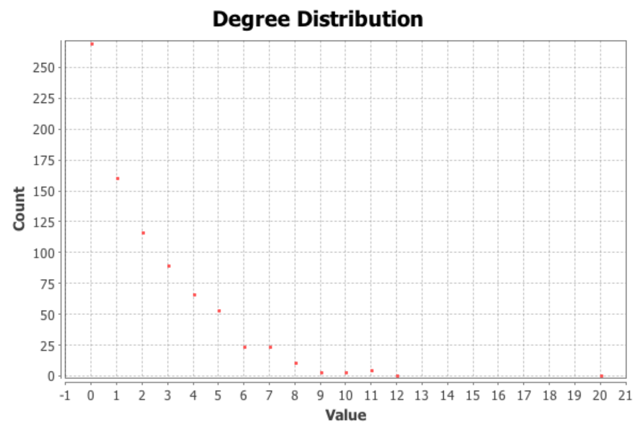
Figure 4.18: Time series of number of nodes online during simulation produced by the *Statistics Component* in Gephi

The robustness of the PolderCast overlay compared to Scribe also seems evident when plotting the time series for the clustering coefficient of the overlay as seen in Figure 4.20(e) and Figure 4.20(f). The average clustering coefficient in PolderCast is not just much higher, but also much more stable. The low clustering coefficient in Scribe might be due to the construction of dissemination trees, as there is less likely to be edges between siblings.

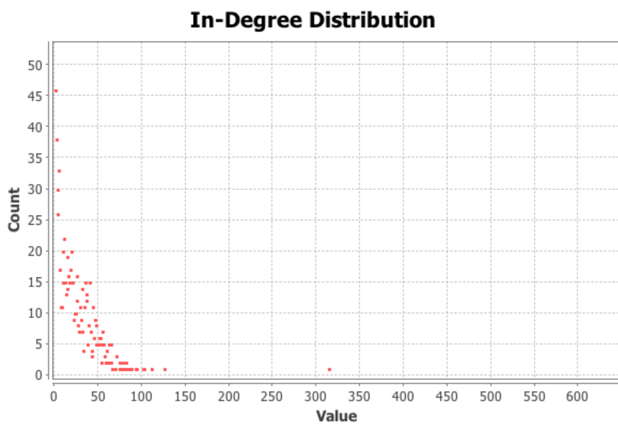
Note that the visual analysis performed in this section is not intended as a comparison of PolderCast and Scribe, but it should serve as an indication of how useful and convenient it can be to perform such a visual comparison of two protocols. Using both the *Statistics Component* in Gephi in order to produce plots, as well as producing visualizations of the overlays leads to several interesting observations which can lead the researcher on a path of further investigations and a more thorough evaluation.



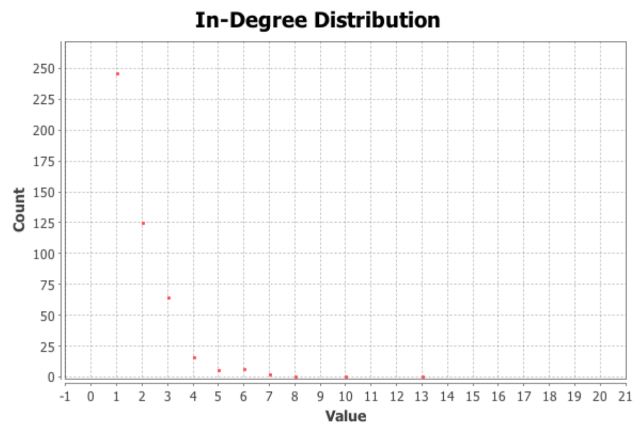
(a) PolderCast



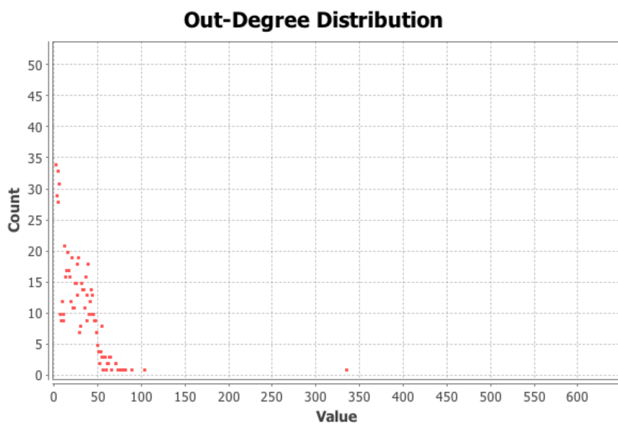
(b) Scribe



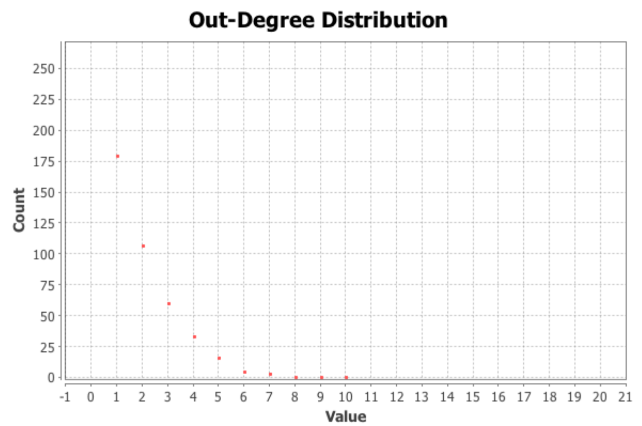
(c) PolderCast



(d) Scribe

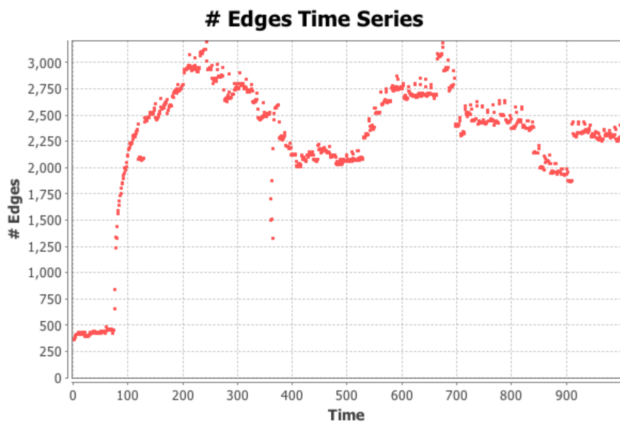


(e) PolderCast

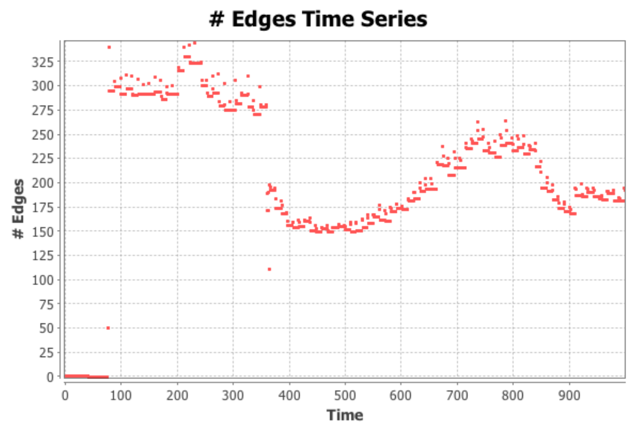


(f) Scribe

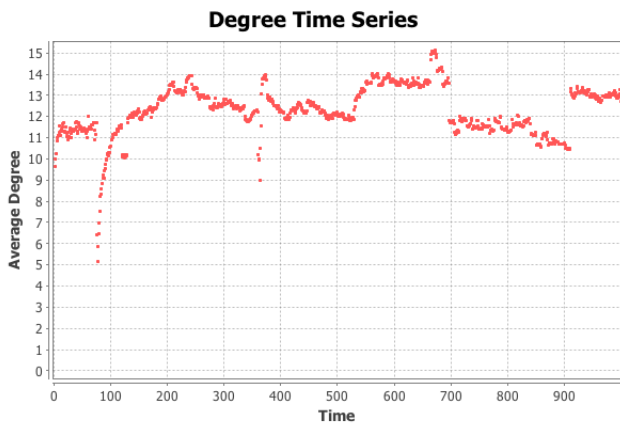
Figure 4.19: Degree distribution of PolderCast and Scribe, produced by the *Statistics Component* in Gephi.



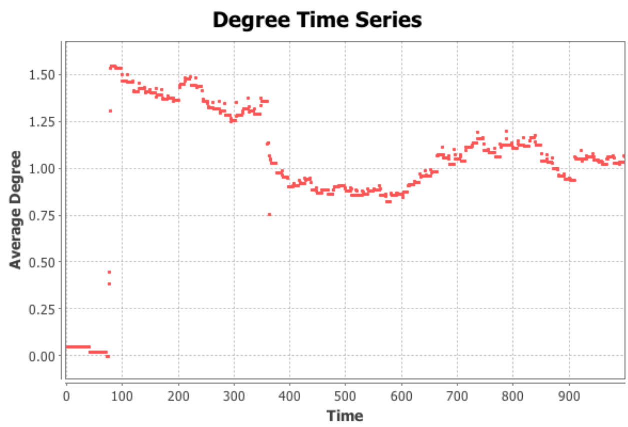
(a) PolderCast



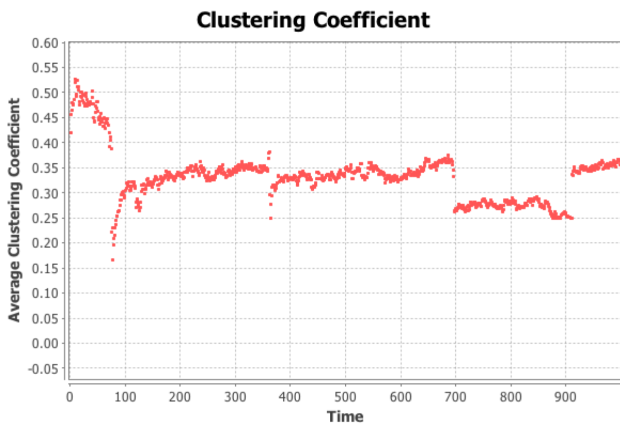
(b) Scribe



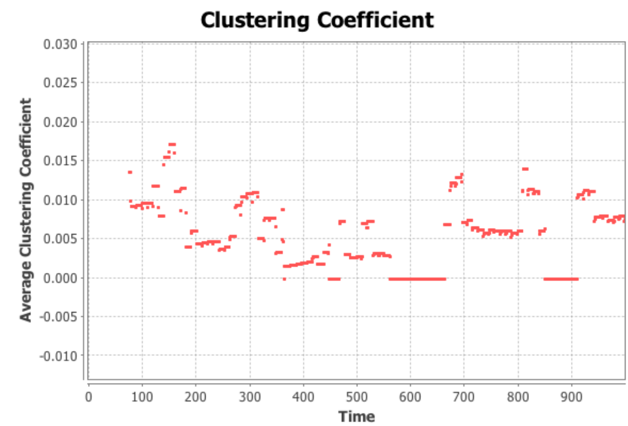
(c) PolderCast



(d) Scribe



(e) PolderCast



(f) Scribe

Figure 4.20: Time series of PolderCast and Scribe, produced by the *Statistics Component* in Gephi.

4.3 Implementation Work

During our implementation work, we encountered several scenarios were VIZPUB proved its usefulness as a tool for both developing and debugging pub/sub protocols. In this section, we will describe some of these experiences, as well as experiences with other aspects of software development and distributed systems research such as using test-driven development and sharing code with the research community.

4.3.1 Using Visualizations to Analyze Datasets

In the beginning of this chapter we mention how we were able to observe a disconnected component in the RINGS layer of PolderCast, as seen in Figure 4.1, and later confirm this was an artefact of the workload being used. More specifically, we used the *Node Query Tool* in Gephi to ensure these nodes had zero overlapping subscriptions with any other node in the overlay by inspecting the "Topics" attribute of the visualized nodes. This was the first practical experience we had with the tool, were its usability was clearly demonstrated. It speaks to how visualizations are also able to provide information of the datasets being used. In the scenario we describe, we used a real-world trace from Twitter [19], which included 1000 user accounts and their subscriptions. However, the PeerNet simulation only included 100 nodes for testing purposes. This lead to a biased sample of the trace, which in this case resulted in a disconnected component. Being able to run simulations with fixed parameters but different datasets, and then inspect the resulting overlay visually is another aspect of using VIZPUB which should prove useful for researcher of topic-based pub/sub systems.

4.3.2 Debugging Pub/Sub Systems Visually

When implementing hit-ratio calculation in the Collector, producing a visualization of the publication message dissemination revealed a bug in the implementation code. In PolderCast, the hit-ratio in the absence of churn should be 100%. However, the Collector would consistently calculate the hit-ratio to 97%. Usually, the first step in debugging such issues is to isolate the problem. In this case, there could be a software bug in the implementation of the pub/sub protocol itself, or the bug might be in the implementation of the hit-ratio calculation. Using VIZPUB, we were able to quickly isolate the problem. By producing a visualization of the publication message dissemination, seen in Figure 4.21, we were able to visually confirm that all nodes that took part in the dissemination received the message. Furthermore, we could confirm this observation by using the Statistics component in Gephi, which enables calculation of the number of connected components in the graph. The calculation revealed that the graph is a single strongly connected component, i.e. from any node in the graph there is a directed path to any other node. This means that every node received the publication. However, the Collector still calculated the hit-ratio to 97%. This indicates that the problem is not with the implementation of the protocol, but with the implementation

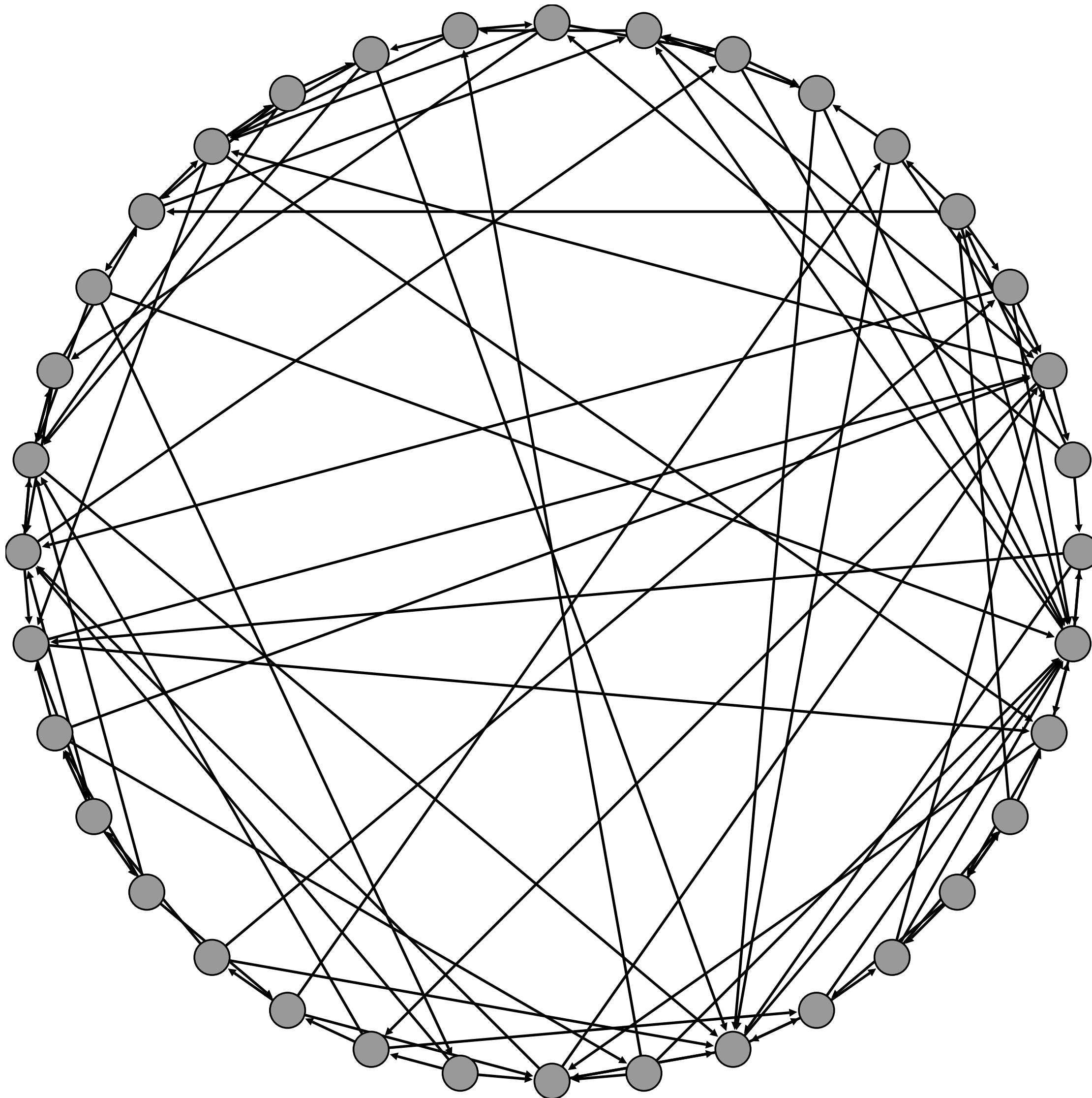


Figure 4.21: Visualization of a fully connected dissemination graph in PolderCast, which revealed a bug in the implementation of hit-ratio calculation

of the hit-ratio calculation. And indeed, by inspecting the hit-ratio calculation in the Collector, we revealed a *off-by-one* software bug, leading to incorrect hit-ratio result.

Debugging is a major part of developing software and arguably the most time consuming, and debugging distributed systems is especially hard. Any tool that aids in debugging is highly valuable to any developer. VIZPUB can aid in both discovering and resolving bugs, as it provides the developer with instant visual feedback of the protocol behaviour.

4.4 Chapter Summary

In this chapter we presented VIZPUB [18], a tool we propose for visualizing the performance of overlay-based pub/sub systems. To the best of our knowledge, this is the first tool of its kind. It is designed to be a generic tool, intended to be used with any pub/sub system. We describe VIZPUB and its system architecture, as well as demonstrate the usage of the tool by providing examples of visualizations. VIZPUB is able to produce two different types of visualizations: (1) visualization of overlay structure, and (2) visualization of publication message dissemination. We also describe our own experiences working with the tool, and how it aided us in debugging as well as in better understanding the characteristics of the different pub/sub protocols in question. We believe our tool is a useful contribution which has value both for researchers, developers as well as students of distributed systems. We share our implementation code by hosting it in a public repository in order to encourage its further development. We presented a poster and held a demonstration of VIZPUB at DEBS 2014 where we were awarded with the price for best poster/demo. We are encouraged by the feedback received from the researchers who visited our demo at DEBS, and we hope our tool will serve as a lasting contribution to the research community.

Chapter 5

Testing and Evaluation

In this chapter, we test the capabilities of VIZPUB as a framework for evalutating pub/sub systems. We extend the evaluation of PolderCast and Scribe found in [30] on a set of topology metrics by using the Statistics API included in the *Gephi Toolkit*.

5.1 VIZPUB as a Framework for Evaluating Pub/Sub Systems

Although the plots seen in Chapter 4 are easy to produce in Gephi, they are not very flexible. For example, it would be useful to be able to superimpose two plots on top of each other in order to effectively compare them. Therefore, in addition to outputting a .gexf file which can be used to produce visualizations and plots in Gephi, The Collector is also able to generate .csv files which can be used to plot a time series of metrics such as degree, clustering coefficient and centralities. Each time point in these time series represents a *Reporter Interval*. Although the Data Laboratory component in Gephi is able to output such .csv files, it is much more convenient to output them directly from the Collector, as opening the Gephi GUI-client for the sole purpose of producing such files manually is much more cumbersome and time consuming, especially on older hardware, or machines without a dedicated graphics card. The Collector is able to calculate various metrics and output them to file by taking advantage of the Gephi Toolkit, which provides an API for the major components of Gephi. Which metrics to output is configurable in the Collector. Currently, the supported metrics which can be exported to .csv files by the Collector include:

- Undirected Degree
- In-Degree
- Out-Degree
- Clustering Coefficient
- Betweenness Centrality

- Closeness Centrality
- Eccentricity Centrality
- Topic Diameter
- Size of Network

This capability of VIZPUB grants researchers who wish to evaluate any pub/sub system immediate access to several metrics. They do not need to reimplement algorithms for the metric calculations. All that is required of them is to implement the *reporter interface*.

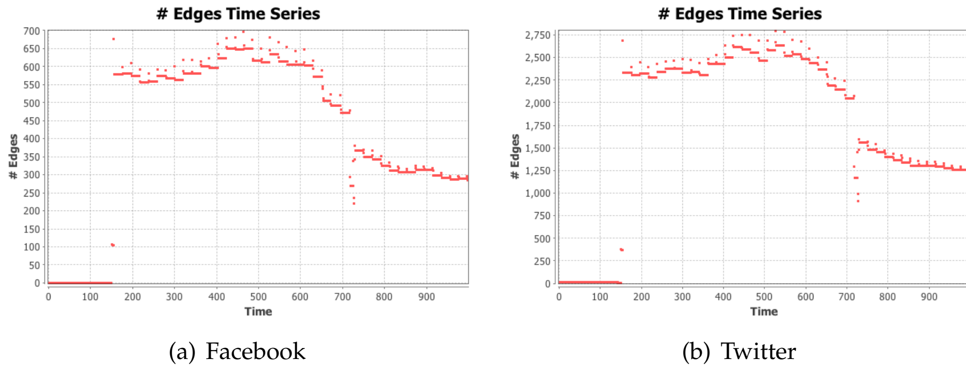
The plots produced in this chapter aim at resembling the plots produced by the Statistics Component in Gephi. However, due to using an external tool for plotting, we can superimpose several plots on top of each other, as well as adjust the format and layout of the plots. Also, it should be noted that the output file format of the Gephi plots are non-vector image files, which is not very suitable for printing. The ability to choose the image format of the plots is another benefit over using the standard plot output of Gephi.

5.2 Experimental Restrictions

As VIZPUB is still in an early prototype state, there are restrictions in terms the total number of nodes we can run as well as the number of reporter intervals. This is due to the file sizes of temporary logs growing linearly with the number of nodes, the number of intervals and perhaps more importantly: the number of edges. This becomes problematic with pub/sub systems such as PolderCast, which generate a high number of edges. The Collector will also consume a lot of memory when calculating custom metrics due to the files note being streamed, but loaded directly into memory. Also, the Gephi Toolkit suffers from performance issues due to an inefficient GEXF-parser. This means that currently, VIZPUB need a high amount of memory and disk space in the order of several Gigabytes in order to operate properly. Unfortunately, although we had access to the needed amount of memory, we were restricted in terms of disk space. For this reason we restrict the number of nodes and reporter intervals in our experiments to 2000 nodes and 1000 reporter intervals.

Another issue is the file sizes growing linearly with the number of publication messages. As the number of publication messages per interval can be in the order of tens of thousands, we evaluate the systems on topology metrics only. Also, due to time constraints we leave out the computational costly *topic diameter* metric, and leave this to future work.

We wish to improve the scalability of VIZPUB in terms of file sizes in the future, otherwise its usefulness for real deployed pub/sub systems would be questionable at best. We describe how to improve the performance of VIZPUB in Chapter 6.



(a) Facebook (b) Twitter

Figure 5.1: Plots describing the difference of number of edges in Scribe when using different subscription workloads.

5.3 Experimental Setup

We run PolderCast and Scribe using the simulation mode in PeerNet. The Simulations consists of 2000 nodes running 1000 PeerNet cycles as well as 1000 reporter intervals. The resulting data points are averages after 5 runs. We use workloads both from Facebook [34] and Twitter [19] in order to model subscriptions. As mentioned in Chapter 4, the Facebook data trace include 3 million user profiles along with 28.3 million friend relations. The Twitter dataset consists of 41.7 million distinct users and 1.47 billion follow/followed relations.

The social relations in Facebook are reciprocal, which leads us to model bidirectional subscriptions. More specifically, a Facebook user is modeled as a topic. The friends list of the particular user profile constitutes its subscription list. All of the entries in this list will include the original user in their own lists of subscriptions. Relationships in Twitter however, are unidirectional. When using the Twitter trace, users are also modeled as topics, but here the list of subscriptions are formed on the basis of the "following" list of the particular user profile. The entries in this list are not required to follow back, therefore subscriptions are also unidirectional.

Churn is based on the Skype super-peer trace from [14], tracing 4000 nodes for 4 weeks, tracking their joining and leaving timestamps. We scale churn to include the first day of this trace in order to not introduce a churn rate which is unrealistically high. For latency between node pairs, we use the King dataset found in [15].

5.4 Results

Figure 5.2 shows average directed degree in PolderCast and Scribe. Averages are calculated by dividing the local value of each node, and divide it by the number of nodes in the overlay for the interval in question, i.e. the average is the *arithmetic mean* for each reporter interval. The averages for each run is then added and divided by the

number of runs. This is the case for all metrics described in this chapter. Figure 5.2 confirms the much higher degree of PolderCast observed in Chapter 4. It also reveals a difference in average degree when using different workloads. In particular, using the Twitter workload seem to cause a higher degree in both PolderCast and Scribe. The difference is most noticeable in Scribe, as the degree is quite low, and thus the discrepancy appears to be larger. By using the *Statistics Component* in Gephi, we are able to produce plots describing the difference in number of edges in Scribe when using workloads from Facebook and Twitter, seen in Figure 5.1(a) and Figure 5.1(b) respectively. It is clear that using the workload from Twitter induces a much higher number of edges. This might be indicative of a higher overlap in subscription interests between nodes, increasing the number of tree structures being constructed. Such an indication of subscription overlap is also mentioned in [30]. Furthermore, the average degree seems to be stable in spite of node churn, which might indicate that the distribution of node degree does not diverge too much from the mean. More specifically, there are few nodes with a very high or a very low node degree. This decreases the likelihood of such a node to leave the network due to churn, which would have a large impact on the average value of node degree. The two plots seen in Figure 5.2 also indicate that there is an even balance between in-degree and out-degree of nodes, as they both have similar values.

Average clustering coefficient is described in Figure 5.3. As expected, PolderCast also has a higher average clustering coefficient. Also here we can observe that the clustering coefficient of PolderCast seems relatively unaffected by using different workloads, while in Scribe the clustering coefficient increases when using the Twitter dataset for subscriptions. Again, this is a natural consequence of the much higher number of edges in Scribe when running the Twitter workload. Also, the fact that PolderCast has a higher clustering coefficient is hardly surprising, as the we know from Chapter 4 that the overlay in PolderCast is more dense.

Results of calculating centrality metrics can be seen in Figure 5.4, Figure 5.5 and Figure 5.6. Again it can be observed in all plots, that the values of the metric calculations in Scribe diverge depending on the what workload is being used for subscriptions. No such discrepancy can be seen in PolderCast.

Figure 5.4 describes the average betweenness centrality of nodes. Although PolderCast have a high betweenness, the values do not seem to fluctuate too much depending on the workload. In Scribe however, we see the mentioned discrepancy. Using the workload from Twitter causes much higher values. Indeed, nodes in the Scribe overlay have the highest average betweenness centralities when running this workload. This could be an indication that Scribe does not scale well in terms of betweenness as the number of edges increases. This could impact the performance of the pub/sub system when running certain workloads.

The average closeness centrality of nodes can be seen in Figure 5.5. Here Scribe populate the top values as using both workloads, although the discrepancy of results depending on the data set used is still higher than in PolderCast. This is to be expected, as closeness is a measure between the distance of a node n and any other nodes in the

system. Seeing that the overlay in PolderCast is much more dense as well as having a shorter diameter, we would expect nodes in PolderCast to have a lower closeness centrality as well.

Figure 5.6 describes the average eccentricity centrality of nodes. The result of running the eccentricity metric seems to be comparable with the results in Figure 5.5. Again, running Scribe has the highest eccentricity when running the Twitter workload. When running the Facebook workload however, Scribe actually has a lower average eccentricity than compared to PolderCast.

The low number of edges and relative high number of disconnected nodes seen in Scribe when running the workload from Facebook might effect the results. It would be interesting to see what is the cause of this. One possible reason might be the scale of the experiment, as we could only run 2000 nodes for the purpose of this thesis. With more time to spare, we would run further experiments at different scales in order to test the effects. However, much work is needed before this is possible, as VIZPUB is still a young tool. We describe such future work in Chapter 6 where we also address some of the limitations of VIZPUB

The low number of edges in Scribe, might also be caused by the amount of churn in the system, as the dissemination structure in Scribe are , in theory, brittle. Indeed, by running a short experiment without churn, we were able to visually confirm that the overlay in Scribe was composed of only one connected component by visually inspecting the graph in Gephi. This might confirm in practice the brittleness of the overlay structures of Scribe, but this issue needs further investigation. This investigation should include adjusting the churn rate in order to see the effects of the overlays, and perhaps compare the density of the graph, looking at the number of edges formed as well as evaluating its connectivity.

5.5 Summary

It should be repeated that VIZPUB is indeed a prototype, and there might still be software bugs which affect the results. But the intention of this chapter lies not only in producing results, but also test VIZPUB as a platform for evaluation of pub/sub systems. Although there are many issues which still needs to be addressed, we believe the potential for such a tool should be explored.

Our main concern when developing VIZPUB was being able to visualize pub/sub systems, but with this chapter we believe we provide some examples to back up the claim that VIZPUB can be used as a tool for evaluation as well as visualization. We believe such a tool would be of great benefit to the research community, and we hope the extendability of Gephi through its ecosystem of plugins can encourage researchers to share more of their implementation code.

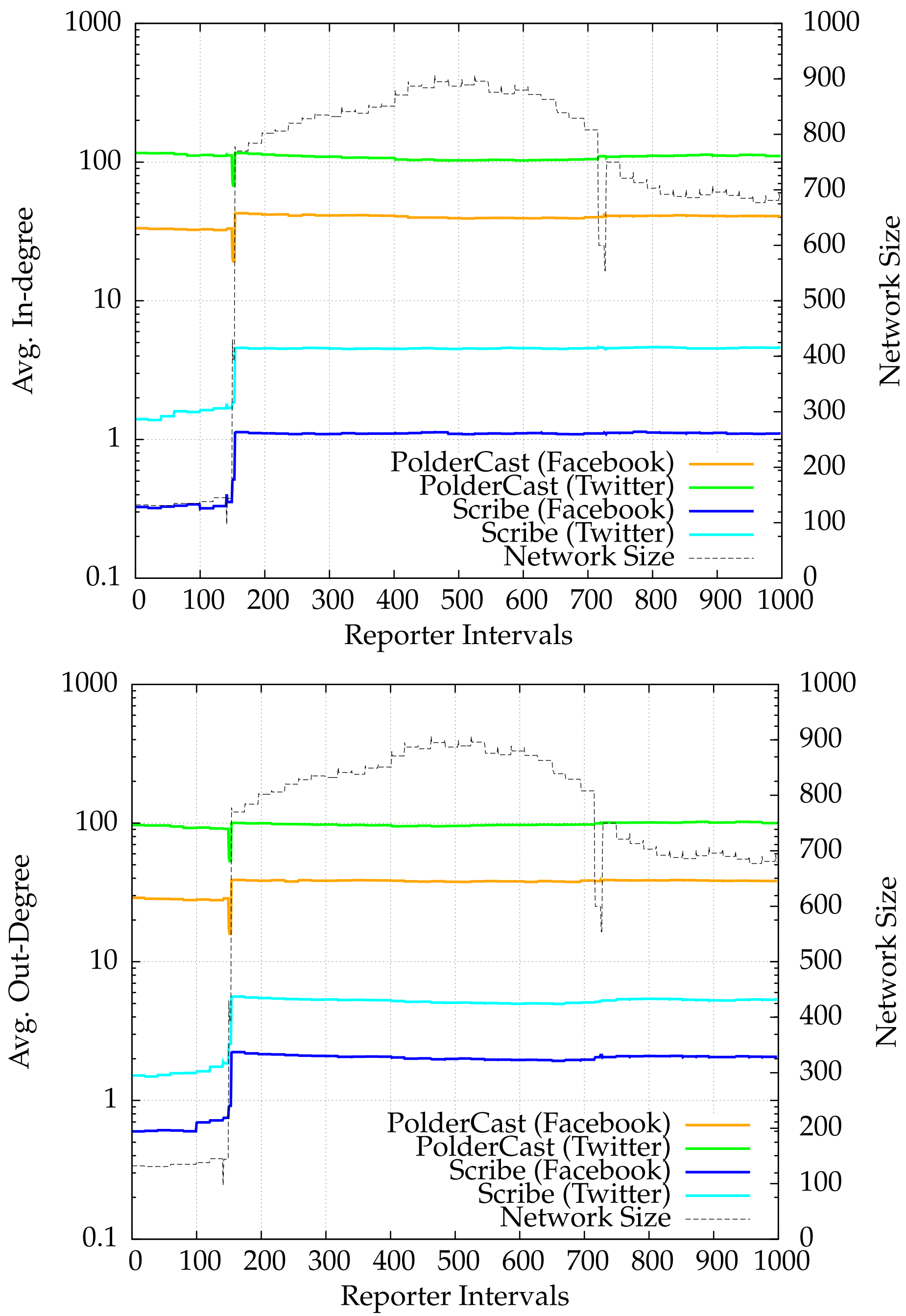


Figure 5.2: Avg. Directed Degrees of PolderCast and Scribe

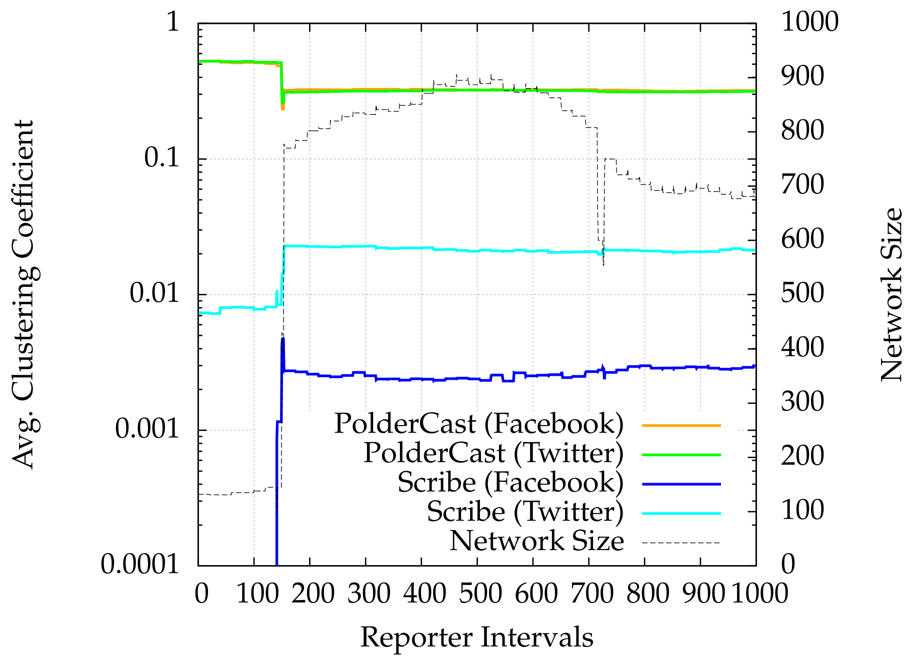


Figure 5.3: Avg. Clustering Coefficient of PolderCast and Scribe

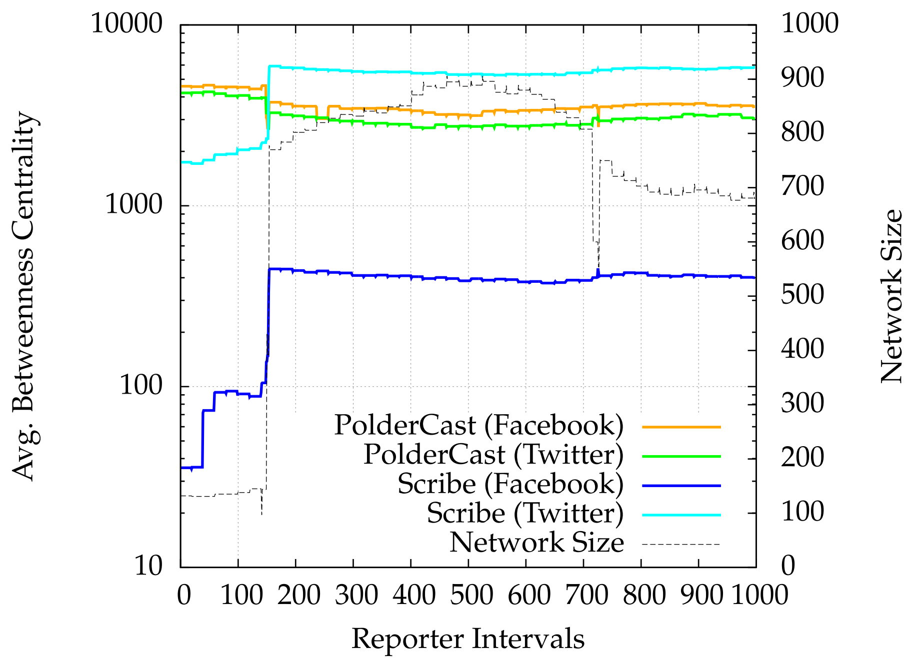


Figure 5.4: Avg. Betweenness Centrality of PolderCast and Scribe

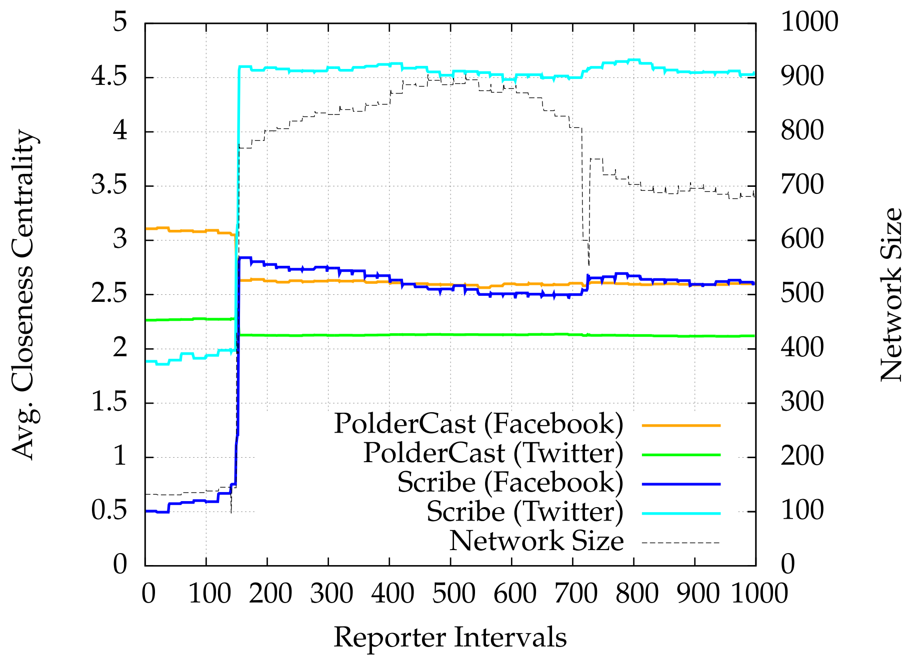


Figure 5.5: Avg. Closeness Centrality of PolderCast and Scribe

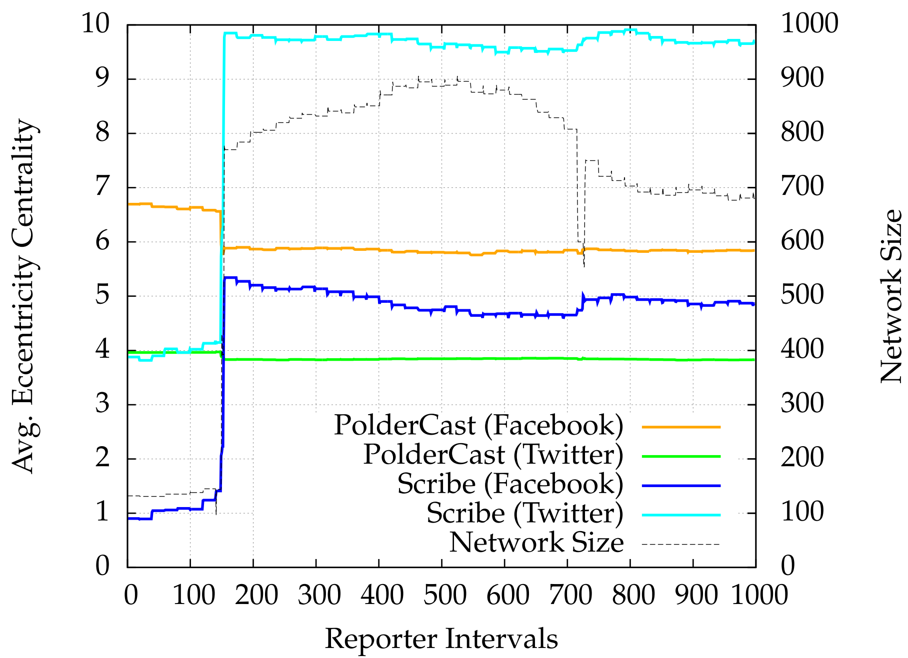


Figure 5.6: Avg. Eccentricity Centrality of PolderCast and Scribe

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we described and demonstrated VIZPUB, a tool we propose for visualizing performance in overlay-based pub/sub systems. We presented a poster and held a live demonstration of VIZPUB at the ACM International Conference of Distributed Event Based Systems (DEBS), held in Mumbai in May 2014, where it was awarded the price for best poster and demo.

To the best of our knowledge, this is the first tool of its kind which enable easy and efficient visual inspection and analysis of pub/sub overlays. The thesis includes several example scenarios where VIZPUB proved to be a useful tool for us in both analyzing as well as debugging pub/sub protocols. Although our main focus is visualizing pub/sub systems, VIZPUB is a generic tool, intended to support any overlay- based system. For this purpose, we define a generic interface, which is the only point of contact between any client code (i.e. overlay-based system) an our framework. The implementation of this interface is the only system specific part of our tool.

We demonstrated our tool further, by providing examples of visualizations. To summarize, VIZPUB supports two different types of visualizations: (1) visualizations of overlay structure and (2) visualizations of publication message dissemination. The former provides a step-by-step visualization of disseminating events, by drawing directed edges which depicts the paths of messages. Also, numeric text labels are applied to nodes which represent the hop count of the publication messages received. We also show how users may also derive the message duplication count by calculating the in-degree of the visualization. We show an example of this type of visualization where we disseminate a single message on a single topic in PolderCast [30]. The example shows the utility of visualizing publication messages disseminations, both for researchers who wants to analyze the dissemination scheme of their protocol, of developers who want to debug the algorithms used for disseminating messages, as well as for students who wants a deep understanding of the particular dissemination process.

We also provided several examples of visualizations which describe the structure

of the overlay, as well as providing examples on how to visualize certain metrics of pub/sub systems through color and text labels. We used these capabilities in order to visually compare two different pub/sub systems, namely PolderCast and Scribe [6]. We also used the capability of the Gephi Open Graph Viz tool [4] in order to produce plots describing the distribution of degree as well as time series of metrics such as number of edges, degree and clustering coefficient.

Finally, we tested the capabilities of VIZPUB as a framework for evaluating pub/sub systems on a set of particular topology metrics. The implementation code for running these metrics are provided by the Gephi framework "for free" through the API provided by the *Gephi Toolkit*. The evaluation revealed the overlay in Scribe changes drastically in terms of number of edges depending on the workload used to model subscriptions. It also seems to indicate that Scribe is more vulnerable to churn than PolderCast, as several nodes are disconnected. However, further evaluation is needed at larger scale in order to investigate these issues properly.

In addition to describing and demonstrating VIZPUB, we also provide a mini-survey, which compares several popular pub/sub protocols both on on the structure of their overlays as well as their dissemination properties. In this comparison we focus on the difficult design decisions researchers face when developing such systems.

We hope our description of our proposed tool and the examples laid out in this thesis makes a convincing case for its utility as a tool for researchers, developers and students alike. The educational benefit of VIZPUB is particularly exciting. Being able to study and experiment with graphs in such a visual and hands-on fashion should provide a very engaging experience for students who are entering the field of distributed systems. We also believe researchers can benefit from the tool, as it provides a deeper insight about the overlay structure as well as the dissemination process. For anyone developing an overlay-based system, whether it implements the pub/sub paradigm or not, VIZPUB can act as a powerful tool for debugging. In this thesis, we provided several example scenarios where we experienced the usefulness of our tool first-hand.

As mentioned, we are not aware of any other proposed tool which is quite comparable in terms of enabling visual analysis of pub/sub systems. And although we imagine numerous possible use cases for VIZPUB, the possibilities of such a tool is still relatively unknown. We hope the research community or any party interested will look at our tool, test it, modify it, extend it and in general find uses for it. During the poster and demonstration of VIZPUB we held at DEBS2014, we were approached by several participant who expressed their interest in using such a tool for their own research. This is why we choose to host our implementation code of VIZPUB in a public repository. We hope the tool will serve as a lasting contribution to the research community, as well as aid in recruiting more students to our field of research.

6.2 Future Work

We believe there is a lot of potential for a tool such as VIZPUB. What we present in this thesis is currently only a prototype, but hopefully it can serve as a starting point for further improvements. In this section we will discuss some of these potential improvements to VIZPUB, as well as some of the issues and limitations we faced in our implementation work.

6.2.1 Further Evaluation

Due to time constraints, we were unable to perform a more extensive evaluation with several configurations. In particular, we would have liked to compare PolderCast and Scribe on a set of dissemination metrics such as *hit-ratio*, *path lengths*, as well as compare the messaging overhead in terms of number of control messages. We expect the result would show that that PolderCast has a higher hit-ratio and lower path lengths at the expense of higher loads. However, we have been unable to show this through an evaluation.

6.2.2 Improving Report File Sizes

Currently, the biggest issue with VIZPUB is scalability in terms of file sizes. Both the Reporter and Collector stores temporary files which can be quite large depending on the scale of the system in terms of number of nodes, edges as well as number of topics and publication messages being posted on each topic. During our experiments we ran into several issues due to restrictions in disk quota on the machine cluster. In particular, a high number of edges cause issues, especially if any attributes are applied to them. But also the number of publication messages will bloat file sizes of the temporary logs considerably. This causes performance issues when the Collector collates these files into the final report. The final report also scales poorly in terms of file size due to the .gexf format being based on XML, which stores data in a declarative text based format. The file sizes of the final report can grow up to several gigabytes, which causes big performance issues when opening these files in Gephi. Due to these performance issues we had to restrict the scale of our experiments. This is unfortunate, and the issues with performance should be addressed quickly in order to increase the scalability of the system. Fortunately, there is a lot of potential in terms of increasing the performance of the system, as it is still in the prototype stage.

6.2.3 Implementing Database Support

Instead of processing log files, it could be useful to take advantage a database for persistence. Graph databases would be an obvious fit for a tool such as VIZPUB. There is already a Gephi plugin for importing and exporting data to such a database, but it is immature and does not support the latest version of the database.[1]. Regardless,

[1]https://marketplace.gephi.org/plugin/neo4j-graph-database-support/

implementing graph database support is most likely a more scalable approach to handling the large amount of data retrieved from the reporters.

6.2.4 Gephi Performance Issues

There are also performance issues with Gephi, as it is still in version 0.8.7. Currently, Gephi is very memory intensive, due to the data model it uses for storing graphs. However, the authors have dealt with this issue in the current development version, where the memory usage has been significantly decreased. However, this version also breaks some features which prevents us from using it. As the development version is still pre-1.0 and rather unstable, we rather choose to wait for it to reach a stable 1.0 version before making any major changes to our tools compatibility. Also, pre-1.0 software usually involves a lot of changes which breaks backwards compatibility. This means users of VIZPUB currently need to use version the 0.8.7 of Gephi until a stable 1.0 version is released.

Another important point to make regarding performance in Gephi is the use of the OpenGL graphics library when rendering graphs. This means that in order to open large graphs in Gephi, it is necessary to use a computer with a dedicated graphics card. The bigger the Graph, the more graphics capability is required.

6.2.5 Reporter Performance Issues

The reporter has some performance issues due to allocating new Java objects each time the Collector pulls data. This causes the Java process to hang, as the garbage collector kicks in to clean up the heap. This is an issue which should be fixed by allocating memory statically and overwriting memory rather re-allocating it. Also, this is a potential bottleneck in the system with regards to memory usage. If the reports are very large, the Java objects might occupy too much memory, causing the Java process to fail. However, this should only be an issue when running simulations at scale, where a process might include several simulated PeerNet nodes but only one reporter.

6.2.6 Visualizing with Custom Colors and Sizes

Most of the visualizations in this thesis use text labels and color in order to convey information regarding the performance of the pub/sub system in question. However, it would also be interesting to visualize metrics using shapes, sizes and custom colors. For example, edges could be of different thickness, according to how many topic attributes were added to them. Also, special nodes such as the *rendezvous* nodes in Scribe could have a custom color to them in order to identify them easily. Alternatively, the "special" nodes could be of a different sizes than "regular" nodes. This is different than the current way of applying colors through the *Statistics Component* in Gephi, as it is dependent on the existence of a node attribute. Any custom color or size would have to be applied by the Collector. However, due to a bug in the external software library[2]

[2] http://https://github.com/francesco-ficarola/gexf4j

used to build the .gexf files, this was not possible to implement.

6.2.7 Collector Scalability

In the current implementation of VIZPUB, the collector is run in a single JVM instance, where the pub/sub nodes connect to it via TCP. There might be issues with the scalability in terms of number of connections, as we have not been able to test the system with more than 30 connections. The Collector also serve as a single point of failure. Ideally, the Collector should itself be a distributed system, with built-in fault tolerance and a replication scheme. This is an important if VIZPUB is ever to be considered *production ready*, i.e. ever being used in real, deployed pub/sub systems. This is not a pressing concern however, as VIZPUB is still a far way from being production ready, but hopefully it will be in the future.

6.2.8 Including Associative Arrays in Gephi

It would be useful to be able to store a map data structure on each graph, i.e. associative arrays. If associative arrays were supported, a user could derive information such as how many duplicate messages a particular node received on a specific topic or single dissemination session. Also, it would enable the user to see how many publication messages a particular node sent or received for a specific topic. Implementing such a structure will most likely involve modifying the source code of Gephi. This is fully possible, since Gephi is an open source product and hosted in a public repository.[3] This would require quite an effort however, as the code base is quite large and it would take time to gain the necessary insight into it.

6.2.9 Implementing Global Attribute Visualization

Currently, it is not possible to visualize graph attributes in Gephi, or indeed list any global attribute in the graph. This would be a useful feature to implement as a plugin. For example, it would be helpful to be able to list all topics on the graph and sort then according to number of subscribers. Our workaround for this is to apply a node label to all nodes which describes a global attribute. Such global attributes include hit-ratio and average control messages sent and received. Visualizing these attributes on every node can be misleading, as it gives the impression that the values are unique to every node. It would be better to display these values in a separate panel, or perhaps even as a large floating text label in the Graph view.

6.2.10 VIZPUB as an Interactive Monitoring System

As mentioned, the visualization of pub/sub systems are done offline, after the execution of the pub/sub system is finished. It is not intended to be a real-time

[3] http://github.com/gephi/gephi

system, where data is pulled and metrics are calculated during system execution. When inspecting the overlay using the *Visualization Unit*, users may delete nodes in order to determine the effect it has on the overlay topology by recalculation such metrics using the *Statistics Component* in Gephi. However, it will not affect any custom metrics which are calculated by the Collector such as control messages sent and received as these metrics are calculated after the execution of the pub/sub system. In order to affect such metrics, the architecture would have to be changed in order to accommodate for a two-way communication, where the Collector would be able to issue commands to the pub/sub nodes. Also, the Collector would have to collect data, calculate metrics and stream this information to Gephi in real-time. Gephi does indeed include an API for streaming data, so it is possible in theory, although in practice there are some issues with this API. This would be incredibly useful however, both as a monitoring system and as an environment for experimenting with protocols and their behavior. For example, imagine being able to right-click a node in Gephi, and ask this node leave the network. The Collector would then issue a leave command to the node in question, and the user could then potentially observe what happens in the Visualization Unit. The interactivity of such a feature would be incredibly engaging and useful both for developers and researchers, as well as students. Such a feature requires extensive re-engineering of VIZPUB, but would be well worth the effort.

6.2.11 Playback issues in the Timeline component

We encountered some issues related to the playback of dynamic graphs in Gephi. The playback is based on time intervals, where the start and end bounds for the interval is encoded as a double. As you play back the graph evolution, these bounds suffer from rounding errors, which might lead to an interval that overlaps to simulation states. These intervals are effectively filter, stripping the graph of nodes and edges that do not belong in the defined interval. However, if the interval crosses the boundary of two different simulation states, the edges and nodes from both states will be included, which often leads to a sudden increase in number of edges during the animation, which abruptly disappear again. However, this issue is easily worked around by defining a sufficiently short interval. We usually set the interval bound to be of length 0.1, as this in our experience never leads to the issues with the time interval overlapping different states. We confirmed this by playing back the graph and controlling the start and end bounds output on the Filter component, as well as visually confirming the absence of any edge animation irregularities. Testing with the Cyclon protocol, we were also able to visually confirm that the number of edges remain consistent for each cycle in the context pane on the top right side of the Gephi GUI.

Bibliography

[1] *An implementation of the Pastry protocol for PeerSim*. http://peersim.sourceforge.net/code/pastry.tar.gz.

[2] S. Baehni, P.T. Eugster and R. Guerraoui. 'Data-aware multicast'. In: *DSN*. 2004.

[3] Roberto Baldoni and Antonino Virgillito. 'Distributed event routing in publish/subscribe communication systems: a survey'. In: *DIS, Universita di Roma La Sapienza, Tech. Rep* (2005), p. 5.

[4] Mathieu Bastian, Sebastien Heymann and Mathieu Jacomy. *Gephi: An Open Source Software for Exploring and Manipulating Networks*. 2009. URL: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154.

[5] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[6] M. Castro, P. Druschel, A.M. Kermarrec and A.I.T. Rowstron. 'Scribe: a large-scale and decentralized application-level multicast infrastructure'. In: *IEEE Journal on Selected Areas in Communications* 20 (2002), pp. 1489–1499.

[7] Chen Chen, Roman Vitenberg and Hans-Arno Jacobsen. 'ElastO: Dynamic, Efficient, and Robust Maintenance of Low Fan-out Overlays for Topic-based Publish/Subscribe under Churn'. In: ().

[8] G. Chockler, R. Melamed, Y. Tock and R. Vitenberg. 'SpiderCast: a scalable interest-aware overlay for topic-based pub/sub communication'. In: *DEBS*. 2007.

[9] The Gephi Consortium. *Gephi Consortium @ONLINE*.

[10] The Gephi Consortium. *GEXF File Format @ONLINE*.

[11] P.T. Eugster, P.A. Felber, R. Guerraoui and A.M. Kermarrec. 'The many faces of publish/subscribe'. In: *ACM Comput. Surv.* 35 (2003), pp. 114–131.

[12] S. Girdzijauskas, G. Chockler, Y. Vigfusson, Y. Tock and R. Melamed. 'Magnet: practical subscription clustering for Internet-scale Pub/Sub'. In: *DEBS*. 2010.

[13] Sarunas Girdzijauskas, Anwitaman Datta and Karl Aberer. 'Oscar: Small-world overlay for realistic key distributions'. In: *Databases, Information Systems, and Peer-to-Peer Computing*. Springer, 2007, pp. 247–258.

[14] S. Guha, N. Daswani and R. Jain. 'An Experimental Study of the Skype Peer-to-Peer VoIP System'. In: *IPTPS*. 2006.

[15] K. P. Gummadi, S. Saroiu and S. D. Gribble. 'King: estimating latency between arbitrary internet end hosts'. In: *SIGCOMM* (2002).

[16] David Janzen and Hossein Saiedian. 'Test-driven development concepts, taxonomy, and future direction'. In: *Computer* 38.9 (2005), pp. 43–50.

[17] Anne-Marie Kermarrec and Peter Triantafillou. 'XL Peer-to-peer Pub/Sub Systems'. In: *ACM Comput. Surv.* (2013).

[18] Nils Peder Korsveien, Vinay Setty and Roman Vitenberg. 'VizPub: visualizing the performance of overlay-based pub/sub systems'. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM. 2014, pp. 326–329.

[19] H. Kwak, C. Lee, H. Park and S. Moon. 'What is Twitter, a social network or a news media?' In: *WWW*. 2010.

[20] H. Liu, V. Ramasubramanian and E. G. Sirer. 'Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews'. In: *IMC*. 2005.

[21] Fotis Loukos, Helen Karatza and Vana Kalogeraki. 'Real-time data dissemination in mobile peer-to-peer networks'. In: *Journal of Systems and Software* (2014).

[22] M. Matos, A. Nunes, R. Oliveira and J. Pereira. 'StAN: exploiting shared interests without disclosing them in gossip-based publish/subscribe'. In: *IPTPS*. 2010.

[23] E Michael Maximilien and Laurie Williams. 'Assessing test-driven development at IBM'. In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE. 2003, pp. 564–569.

[24] Stanley Milgram. 'The small world problem'. In: *Psychology today* 2.1 (1967), pp. 60–67.

[25] A. Montresor and M. Jelasity. 'PeerSim: A Scalable P2P Simulator'. In: *P2P Computing*. 2009.

[26] Mark EJ Newman. 'A measure of betweenness centrality based on random walks'. In: *Social networks* 27.1 (2005), pp. 39–54.

[27] F. Rahimian, S. Girdzijauskas, A.H. Payberah and S. Haridi. 'Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe Enabling Rendezvous Routing in Unstructured Overlay Networks'. In: *IPDPS*. 2011.

[28] J. Reumann. *GooPS: Pub/Sub at Google*. Lecture & Personal Communications at EuroSys & CANOE Summer School, 2009. Oslo, Norway.

[29] A.I.T. Rowstron and P. Druschel. 'Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems'. In: ed. by R. Guerraoui. Vol. 2218. LNCS. Springer, Berlin/Heidelberg, 2001, pp. 329–350.

[30] Vinay Setty, Maarten van Steen, Roman Vitenberg and Spyros Voulgaris. 'PolderCast: fast, robust, and scalable architecture for P2P topic-based pub/sub'. In: *Proceedings of the 13th International Middleware Conference*. Middleware '12. 2012.

[31] *Tibco rendezvous*. http://www.tibco.com.

[32] S. Voulgaris, D. Gavidia and M. van Steen. 'CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays'. In: *Journal of Network and Systems Management* 13 (2 2005), pp. 197–217. ISSN: 1064-7570. URL: http://dx.doi.org/10.1007/s10922-005-4441-x.

[33] S. Voulgaris and M. van Steen. 'Hybrid dissemination: adding determinism to probabilistic multicasting in large-scale P2P systems'. In: *Middleware*. Ed. by R. Cerqueira and R. Campbell. LNCS. Springer, New York, 2007, pp. 389–409.

[34] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy and B. Y. Zhao. 'User Interactions in Social Networks and their Implications'. In: *EuroSys*. 2009.

[35] B. Wong and S. Guha. 'Quasar: a probabilistic publish-subscribe system for social networks'. In: *IPTPS*. 2008.

[36] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz and J.D. Kubiatowicz. 'Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination'. In: *NOSSDAV*. 2001.