Demo: VizPub: Visualizing the Performance of Overlay-Based Pub/Sub Systems

Nils Peder Korsveien  
University of Oslo, Norway  
nilspk@ifi.uio.no

Vinay Setty  
University of Oslo, Norway  
vinay@ifi.uio.no

Roman Vitenberg  
University of Oslo, Norway  
romanvi@ifi.uio.no

ABSTRACT

We propose a tool for visualizing a variety of performance metrics in topic-based publish/subscribe systems, ranging from dissemination of publications to overlay properties. The tool can be used for gaining insight into the system performance and for comparing different pub/sub systems.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed applications

Keywords

Pub/Sub Systems, Visualization, Peer-to-Peer

1. INTRODUCTION

Pub/sub systems are characterized by a large number of important performance metrics related to delivery latency, bandwidth consumption, communication and computation overhead, load distribution, and fault-tolerance. Many of these aspects are known to be at stake with each other [5, 10]. Providing a suitable balance in performance is a key design challenge for pub/sub.

Overlays play an important role in pub/sub architectures, both in data centers [4, 8, 9] and large-scale P2P settings [1, 3, 10]. The use of overlays necessitates considering the overhead of overlay maintenance as well as structural overlay properties, such as fanout, clustering, and connectivity. These properties further enlarge the set of performance-critical aspects in pub/sub that require careful study and attention.

We propose a tool for visualizing the execution of any given distributed pub/sub system step by step, with respect to the performance metrics. Each node in the system under analysis needs to log a number of local parameters at selected reporting intervals. The tool collects these logs to a single site and computes the global metrics of performance at a system-wide scale. Then, our tool allows the user to replay the execution using the Gephi framework [2], while visualizing the overlay structure, communication between the nodes, and associated performance metrics at each selected interval. In particular, the overlay structure can be visualized on a per-topic basis (i.e., including only the nodes interested in a given topic) or globally for all the nodes. The tool can be used both for data center and P2P overlays. Even though our main goal is visualizing the performance of pub/sub systems, the tool can be used for visualizing other types of overlay-based systems that disseminate data, such as [6].

The main benefit of using the proposed tool is the ability to gain deeper insight, e.g., about the overlay structure and the publication process. For example, how easily can the overlay be disconnected by failing just a few nodes? How strongly clustered the overlay is? Is there a topic for which the dissemination is relatively slower, or slower than a certain absolute threshold? Is the dissemination slower or less reliable for a particular source node or overlay neighborhood? Are there many duplicate messages received?

Such an insight allows for identifying potential weaknesses or deployment anomalies of a given pub/sub system. For example, we experimented with the PolderCast [10] system, which encompasses several levels of overlays. With the help of our visualization tool, we immediately saw that three nodes were disconnected from the others in the higher level overlay. Initially, we suspected a software bug. Then, we verified that this situation was caused by an artifact in the input workload where the three nodes had zero overlap of interests with any other node in the system. In this case, the nodes should still be well connected in the lower level overlay of PolderCast, which we successfully verified via our tool. To the best of our knowledge, no other available tool would facilitate such an easy detection and validation.

Another benefit of the tool is that it endows the user with the ability to effectively compare the performance of different pub/sub systems and protocols. Specifically, the user can run different systems for the same input settings (same real or simulated network with the same number of nodes) and on the same workload (publications and subscriptions). Each node in each run will be recording local information as explained above. Then, by using the tool, the user could replay multiple executions and compare the performance at selected points in time.

Observe that the process used by the tool is generic and it can be applied to any pub/sub system in existence. It should be mentioned that despite the significant diversity in system design, the performance metrics of interest are mostly identical for all pub/sub systems (see [5, 10] for a detailed discussion). This allows us to define a generic list of performance metrics and a generic reporting interface. The only system-specific part in the process is the implementation of the reporting interface. In this demo, we implement such an interface for the PeerSim simulation framework [7]. In fact, we implemented the interface both for the simulation and real deployment modes of PeerSim. Since the implementation of the re-
2. VISUALIZED PERFORMANCE METRICS

Our proposed interface for reporting local node information is designed to support most common metrics important for performance of pub/sub systems, as outlined in [5, 10]. The supported metrics can be divided into three categories: (a) structural overlay properties such as the node degree, diameter, and clustering coefficient, (b) dissemination properties including hit ratio, path lengths, and number of duplicate messages received, and (c) communication overhead of maintaining the overlay, measured in either bytes or the number of control messages.

The Gephi framework allows the user to visualize graphs, which we exploit towards visually presenting pub/sub overlays. In addition to depicting individual nodes and edges, both nodes and edges in Gephi may have attributes attached to them. Those attributes can be visualized as numbers as well as by color (see the snapshots in Section 4). It is possible to show all nodes at once or filter them according to an attribute, e.g., by topic. Furthermore, statistical properties of the attributes such as average degree can be computed across all the nodes, for a particular topic, or for an individual sender node.

We are able to visualize both instantaneous metrics (e.g., diameter at a particular point in time) and aggregate metrics over time. Aggregate values can be computed over time windows of configurable duration. For example, we support computing the average communication overhead both from the beginning of the execution or only between the last two consecutive reporting intervals.

3. SYSTEM ARCHITECTURE

In this section, we describe the architecture and implementation details of VizPUB. The goal of VizPUB is to visualize the overlay and various performance metrics (described in Section 2) for a given pub/sub system driving a given workload. To achieve this, in our system, there are three main components (see Figure 1): (1) Reporter, (2) Collector and (3) Visualization Unit. In a nutshell, VizPUB collects raw information needed for various performance metrics from individual nodes participating in the pub/sub overlay at each reporting interval via the “Reporter”, then the reported information is collected and aggregated by the “Collector” and translated into the format understood by the “Visualization Unit”. Finally, the Visualization Unit computes some of the graph related metrics and visualizes them via “Gephi” framework. This flow of information is depicted by the arrows in the Figure 1.

Figure 1: Architecture diagram of VizPUB

We provide a number of snapshots for the PolderCast system that illustrate the communication overhead and dissemination process. While it is possible to visualize the results for the other systems listed above, we could not include them due to the space constraints in this short description of the demo.

### Table 1: Reporter Interface Methods

<table>
<thead>
<tr>
<th>Method Name and Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long reportId()</td>
<td>The unique id of this node</td>
</tr>
<tr>
<td>long[] reportNeighborIds()</td>
<td>The ids of this node’s neighbors</td>
</tr>
<tr>
<td>long[] reportTopics()</td>
<td>List of topic ids this node subscribes to</td>
</tr>
<tr>
<td>long reportControlMsgsReceived()</td>
<td>Number of overlay control messages received</td>
</tr>
<tr>
<td>long reportControlMsgsSent()</td>
<td>Number of overlay control messages sent</td>
</tr>
<tr>
<td>long reportControlBytesReceived()</td>
<td>Number of overlay control bytes received</td>
</tr>
<tr>
<td>long reportControlBytesSent()</td>
<td>Number of overlay control bytes sent</td>
</tr>
<tr>
<td>PubMessage[] reportPubMsgsReceived()</td>
<td>Reports list of publication messages received</td>
</tr>
<tr>
<td>PubMessage[] reportPubMsgsSent()</td>
<td>Reports list of publication messages sent</td>
</tr>
</tbody>
</table>

In the data flow described above, the Reporter works in online mode reporting the required information at the end of each reporting interval. The Collector is designed to collect the reported information in online mode but the computation, aggregation and derivation of various metrics can be done in offline mode. Finally, the Visualization Unit always works in offline mode. Once VizPUB is configured manually for a specific pub/sub system with specific parameters, the collection and aggregation of the visualized metrics requires no human intervention.

Note that VizPUB works both for simulation mode and real distributed deployment mode of the pub/sub implementation without any further modifications. This is attributed to the fact that the Collector and the Visualization Unit is independent of reporting. The only deployment specific parameters will be in the implementation of Reporter interface. For example, VizPUB configured to visualize performance metrics of PolderCast on PeerSim framework works both in simulation mode as well as in a real deployment.

### Table 2: Data Structure of a Publication Message

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long MsgId</td>
<td>Unique id of the this message</td>
</tr>
<tr>
<td>long TopicId</td>
<td>Topic id for which this message was generated</td>
</tr>
<tr>
<td>long SourceId</td>
<td>Id of the previous hop node</td>
</tr>
<tr>
<td>long DestinationIds[]</td>
<td>Ids of the next hop nodes</td>
</tr>
<tr>
<td>long OriginalSenderId</td>
<td>Node id of the message source</td>
</tr>
<tr>
<td>long TimeStamp</td>
<td>Timestamp of the message sent/received</td>
</tr>
</tbody>
</table>

In the data flow described above, the Reporter works in online mode reporting the required information at the end of each reporting interval. The Collector is designed to collect the reported information in online mode but the computation, aggregation and derivation of various metrics can be done in offline mode. Finally, the Visualization Unit always works in offline mode. Once VizPUB is configured manually for a specific pub/sub system with specific parameters, the collection and aggregation of the visualized metrics requires no human intervention.

Note that VizPUB works both for simulation mode and real distributed deployment mode of the pub/sub implementation without any further modifications. This is attributed to the fact that the Collector and the Visualization Unit is independent of reporting. The only deployment specific parameters will be in the implementation of Reporter interface. For example, VizPUB configured to visualize performance metrics of PolderCast on PeerSim framework works both in simulation mode as well as in a real deployment.

3.1 Reporter

The goal of the Reporter is to capture the various performance metrics of the pub/sub system over a period of time. To achieve this, we provide a reporting interface, which can be implemented by the Reporter at each individual nodes participating in the pub/sub system (see Table 1). By implementing this interface the nodes can report a number of local parameters using their local knowledge at each reporting interval via the Reporter. The reporting happens at the end of each reporting interval. All the intermediate information and bookkeeping for reporting is done by Reporter since it has access to all the information of pub/sub protocols. At the end of each reporting interval, the Collector invokes the Reporter interface and pulls the information returned by the interface.

It is not hard to see that all the metrics mentioned in Section 2 can be derived by collecting the information reported from each node via the reporting interface listed in Table 1.

The structural overlay properties such as the degree, diameter and clustering coefficient can be derived by building a topology of...
the overlay (nodes and edges) using the information reported via `reportId()` and `reportNeighborIds()`.

The dissemination properties including hit ratio, path lengths, and number of duplicate messages received, can be derived by analyzing the list of publication messages (`PubMessage` structure defined in Table 2) reported via `reportPubMsgsSent()` and `reportPubMsgsReceived()`. For example, to compute the hit ratio w.r.t a specific topic, we need to know the total number of subscribers of that topic and the number of subscribers who received a specific publication message generated for that topic. The former can be derived by analyzing all the topics subscribed by each node using `reportTopics()`. The latter can be computed by counting the number of subscriber nodes who received a specific publication message generated for a given topic reported via `reportPubMsgsReceived()`. Similarly, the dissemination path length in the sub-overlay of a specific topic w.r.t a given source can be computed by ordering the publication messages sent/received using `TimeStamp` attribute of the publication messages as reported by individual nodes. The number of duplicate messages is derived by counting duplicates of each unique publication received at each node. Finally, the communication overhead in terms of both number of control messages and their bandwidth consumption can be computed for any given reporting interval using the methods: `reportControlMsgsSent()`, `reportControlMsgsReceived()`, `reportControlBytesReceived()` and `reportControlBytesSent()`.

Note that the Reporters can be configured to report at a chosen reporting interval. In addition, to improve online performance, the reporters can choose to report only partial information depending on the required metrics to be visualized. For example, to visualize the dissemination metrics for a specific topic, only the publication messages generated for that topic can be reported. This requires the user to know the visualized metrics but does not necessitate any changes to the Collector or Visualization Unit.

3.2 Collector

The Collector is responsible for pulling the information from the Reporters and aggregating it into various visualized metrics. Some graph related metrics such as degree, clustering coefficient and diameter can be internally computed by Gephi framework for a given overlay. To achieve this, the Collector employs .gexf file format\(^1\), which is interpreted by the Gephi framework for computing and visualizing graph related metrics. To generate the .gexf files, the Collector creates the nodes in the graph from the unique ids reported by each node and the edges in the graph are derived from the neighbor ids reported by each node.

The Visualization Unit described later in this section can interpret both instantaneous metrics (current values) and aggregated metrics (based on historical information). The instantaneous metrics are directly computed by the Gephi framework without any additional computations by the Collector. On the other hand aggregated metrics are computed by the Collector based on the relevant historical information from the sequence of reports of configurable length or time range. Examples of aggregated metrics include average number of overlay control messages sent/received, hit ratio, dissemination path lengths and duplicate messages. To support these metrics, the Collector introduces a number of custom attributes for each node in the .gexf file format. The Collector combines all reports into a single .gexf file, which is interpreted by the Visualization Unit later when replaying the execution.

Note that the Collector is designed to aggregate the selected metrics in offline mode. This avoids the Collector becoming a bottleneck in processing incoming information from the Reporters. Alternatively, the Reporters can log all the information locally and push it to the Collector at the end of pub/sub execution without affecting the final outcome.

3.3 Visualization Unit

The Visualization Unit is a machine running the Gephi framework. The final .gexf file generated by the Collector can be opened in Gephi and the user can choose to playback the pub/sub execution. The user can filter the visualization in various ways: the sub-overlay metrics of all the subscribers of a specific topic or by specifying range of values for the custom attributes etc.

In Figure 2 we show a snapshot of the visualization unit, i.e. the Gephi User Interface, and some of its useful components. The bottom part consists of the Timeline Component which enables the user to play back the simulation, animating both size, color and labels of both nodes and edges. The timeline represents the entire time range of the simulation, where a Time Window can be adjusted in order to filter out any node or edge that does not exist within the specified interval. The user is able to manipulate the time window by defining its upper and lower bounds. Playback of the simulation is possible through defining a step size and a playback speed in addition to choosing an appropriate time window size. The user may then initiate playback by pressing the Play/Pause Button. This results in the time window iterating through the entire timeline range,

---

\(^1\)GEXF (Graph Exchange XML Format) is a language for describing complex networks: http://gexf.net/format/
where playback speed is the delay between each iteration, and step size is the length which the window is moved. The user may pause playback at any time, or jump to another time point by defining new time window bounds. Using the Statistics Component, the user may then calculate selected metrics for the chosen time interval.

Using the Node Query Tool, the user may click on any node in the graph in order to display the node query panel found on the top left. The node query panel lists the node properties such as size and color, as well as the attributes of the nodes. These attributes include any custom attributes which were added by the Collector such as “Topics” or “Subscription Size”, in addition to default attributes such as “Time Interval”. The user is able to both inspect and edit the values of these attributes. For example, the user may change the time interval values of a node, in order to make this node appear or disappear at a specific point in time. The effect this has on particular properties of the overlay topology could then be determined by recalculating the appropriate metrics.

4. DEMONSTRATION DESCRIPTION

In this section, we demonstrate our tool by providing visualizations which are produced by running PolderCast simulations using PeerSim, VizPUB and the Gephi framework (version 0.8.2). We choose not to include visualizations from any other protocol due to space restrictions. The scale of the visualization in terms of number of nodes and edges have been restricted in order to make them readable. We have been able to produce these visualizations at larger scale with thousands of nodes and edges in total using a personal laptop. In a potential demo, we would be able to present visualizations at a similar scale.

In Section 1 we described a scenario where we observed three disconnected nodes when experimenting with the PolderCast system. Figure 3 represents the overlay we observed in this scenario. This visualization consists of 29 nodes and 315 edges, after 100 reporting intervals as well as 100 PeerSim cycles. The three disconnected nodes are clearly visible to the right. As mentioned, we were able to confirm that this topology is an expected result of the PolderCast protocol with the particular dataset being used. Visualizations of overlay topologies are useful in order to experiment with such edge cases.

In Figures 4(a) and 4(b), we visualize maintenance messaging overhead in the PolderCast system. These visualizations represent the higher level overlay in the PolderCast system after 100 reporting intervals and 100 PeerSim cycles, with 30 nodes and a total of 280 edges. In Figure 4(a), each node label represents the average number of gossip messages sent per node, while in Figure 4(b) the labels describe the average number of gossip messages received per node across all three protocols of PolderCast. The averages are calculated by dividing the total number of gossip messages sent or received per node during system execution by the total number of reporting intervals. It is possible to infer a lot of information based on these visualizations. For example, it is trivial to identify overloaded nodes by their deep red color, as a gradient is applied to the nodes based on their label values. The visualizations also indicate that these overloaded nodes have a higher undirected degree compared to the other nodes in the system. Furthermore, by comparing the labels across the two visualizations, it is possible to see that nodes receive almost twice the number of messages than they send.

In Figures 4(c) and 4(d) we visualize publication message dissemination in PolderCast. We publish a single message on a topic overlay consisting of 17 nodes, and build directed edges tracing the path of the message through the system. In Figure 4(d), the node labels represent how many duplicate messages a node received during dissemination, where 0 indicates that no duplicate messages were received. In Figure 4(c) the node label represents the hop count of the message when a node received it. Based on these visualizations, it is possible to identify communities which received the message later than others, or identify nodes who receive a high number of duplicate messages.

5. REFERENCES