Bowlmap: Network Monitoring and Debugging through Measurement Visualization

Benjamin Vahl*, Francisco H. Luque†, Thomas Hühn*, Cigdem Sengul*
*Telekom Innovation Laboratories at TU-Berlin
Email:{bvahl,thomas,cigdem}@net.t-labs.tu-berlin.de
†Universidad Carlos III de Madrid (UC3M)
Email: fcojose.herrera@alumnos.uc3m.es

***Abstract*—In this paper, we present Bowlmap, a web-based visualization tool designed for network monitoring and debugging. Bowlmap offers a highly-customizable representation of different measurement traces in the same network map. An important feature of Bowlmap is that it provides the user a convenient way to visualize both live and offline measurements and combine results from different sources. In addition to its flexibility, Bowlmap is designed to achieve low bandwidth usage, which allows faster visualization updates to the viewers. We demonstrate Bowlmap's capabilities and performance through several case studies to show how Bowlmap facilitates network monitoring and debugging.**

***Index Terms*—Visualization, network debugging, monitoring.**

I. Introduction

Monitoring and debugging a network, composed of several components, is a challenging task. Network researchers typically rely on measurements performed by tools such as *tcpdump* that capture traffic. However, as the number of nodes and measurement sources increases, identifying and combining the relevant information becomes a daunting task. In these cases, visualization can significantly improve our understanding of networked systems. Especially, *real-time* visualization of data allows perceiving complex relationships between different parts of the system, and quickly ascertain whether the results match expectations and also, identify unexpected results. Such visualization may also help combine several quantitative results to see the spatio-temporal relationships.

In this paper, we present such a visualization framework, designed - initially - to visualize the BOWL (Berlin Open Wireless Lab) wireless network at Technische Universität Berlin (TUB), which is an outdoor WiFi network of 46 nodes used both for Internet access and as a testbed for wireless research [1], [2]. With Bowlmap, we achieve live network map support that combines different measurement sources (e.g., node states, connectivity, routing state etc.) in an intuitive manner, which allows us to monitor the changes to the network and debug problems more easily. Given the fact that testbeds undergo continuous changes, we extended our design to create a modular and flexible visualization framework that can handle these changes as well as completely new network set-ups.

In our framework, there are four main components: (1) measurement sources, (2) measurement post-processing, (3) a state server and (4) a client (see Fig. 1). The measurement sources are, for instance, tools running in the network and writing to a database back-end. The measurement post-processing component collects information from the different sources, and communicates them to a state server through an XML-RPC-based interface. Through this interface, the state server gathers all the new information about what and how to visualize the measurements. The state-server also provides an interface for client queries and transports a compact per-client map object, which carries only the updates since the last query. This way, the bandwidth necessary to send the updates is reduced, allowing faster updates. Finally, the web frontend at the client side periodically queries the state server and processes the received map object to generate a visualization. It is built using OpenLayers which is an open source Javascript framework to embed dynamic maps into web pages.

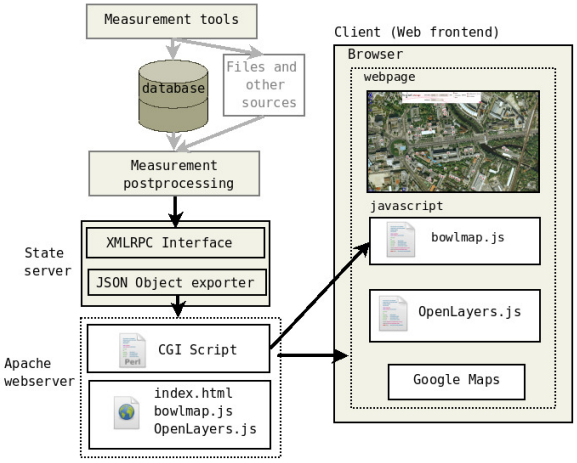


Fig. 1. Bowlmap architecture.The arrows show the information flow from measurement tools to the client browser. The black arrows represent the information flow inside the Bowlmap architecture, whereas the gray flows represent input to the Bowlmap.

Our design achieves the goal of flexibility by handling purely measurement-related actions in separate components (the measurement sources and measurement postprocessing). Hence, no change is necessary to the network state server or web frontend when there are modifications to measurement sources, or when new measurement sources are added. Furthermore, addition of new measurement sources do not require building everything from scratch and can be incrementally

978-1-4673-1239-4/12/$31.00 ©2012 IEEE

added to the existing system. We evaluate the flexibility and scalability of Bowlmap through several examples from the BOWL network, as well as based on traces from other networks.

The rest of the paper is organized as follows. In Section II, we describe the Bowlmap design. In Section III, we present several examples that show the representation ability, flexibility and scalability of Bowlmap. Section IV presents the related work and we conclude with future work in Section V.

II. Bowlmap Design

Bowlmap [1] exports a browser-based front-end using JavaScript, which allows users to view visualization of their target network remotely and from a range of devices such as hand-helds. Nowadays, each major browser technically supports interactive applications and hence, a JavaScript program works sufficiently consistently and efficiently on modern browsers. However, a challenge with implementing the Bowlmap as a browser-based application is the requirement of sending visualization data over the network. In order to scale, the amount of information that needs to be communicated to clients for visualization must be kept low. The work presented in this paper is motivated by this challenge. In the rest of this section, we explain the design of Bowlmap and how it achieves flexibility and scalability in more detail.

A. Higher Flexibility with Postprocessing Plug-ins

In Bowlmap, we use measurement postprocessing scripts to be able to modify or update visualizations easily (see Fig. 1). These scripts read measurement sources and invoke XML-RPC functions to draw the current network state, as configured by the user, on the map. For instance, let's assume we would like to monitor node on and off states in the network and depict this information by putting green icons for nodes in "on" state, and red icons for nodes in "off" state in their respective locations on the network map. Here, the measurement sources provide node locations, and on and off information to the measurement postprocessing script. Then, this script communicates the updates about which nodes should be represented as on or off to the state server. The state server maintains the current network state, and upon receiving client queries, transmits this information.

Postprocessing scripts provide high flexibility to developers as they can be written in any language with an XML-RPC library support. The main functionality of these scripts include connecting to servers and databases (or accessing files) and querying databases (or reading files). Combining these with the state server XML-RPC functions provided by the Bowlmap framework, it becomes possible to write scripts that serve different visualization needs. Fig. 2 shows an example. Following our previous example, this Python script adds nodes to a Google maps layer, which can be in two states: "on" or "off". The script first lays the background as Google maps satellite view (i.e., "set_gmaps_layer"), then scopes the view to the current network range (i.e., "set_map_view"), and finally specifies the underlying vector layer (i.e., "set_vector_layer",

```
1
#Comment: Set state server & database connection
stateserver= xmlrpclib.ServerProxy([URL])                    3
query = [SQL query node location & state]
db_connection = psycopg2.connect([conn_string])              5

#Comment: initialize map view and layers                     7
stateserver.set_gmaps_layer('gsat', 'Google S',
                            'G_SATELLITE_MAP')               9
stateserver.set_map_view([x1],[y1],[x2],[y2])
stateserver.set_vector_layer('nodes', 'Nodes')               11

#Comment: icons for nodes                                    13
stateserver.set_image('node_on',[image_URL], 'on')
stateserver.set_image('node_off',[image_URL], 'off')         15

#Comment: try querying the database                          17
while main_loop:
  try:                                                       19
    result = mpslib.query(db_connection, query)
  except: [error handling code]                              21

#Comment: Go over query result                               23
  for index in range(len(result)):
    row = result[index]                                      25
    node = row['node']
    coords = [float(obj['lon']), float(obj['lat'])]          27
    if  (int(obj['state']) == 1):
      image = 'node_on'                                      29
    else:
      image = 'node_off'                                     31
#Comment: add node with image and coords to nodes
    layer
    stateserver.set_icon(node,node,image,20,'nodes',         33
        coords)
```

Fig. 2. Example code listing for adding nodes to a map.

"nodes" layer) (Lines 7-11) [1] The script next assigns two different images to represent the "on" and "off" states (Lines 12-15). Then, in the main loop, it attempts to connect and query the database (Lines 17-23) and if successful, for each row in the query result, the corresponding icon based on the node state is set on the map (Lines 23-33).

B. Reducing Bandwidth Use via Incremental Updates

As explained in the previous section, the measurement postprocessing scripts update the map information using an XML-RPC interface to the state server. In our implementation, the state server maintains this information as a directed acyclic graph (DAG), where each node of the DAG is a part of the map description, timestamped based on its update time. To reduce the communication overhead, the state server keeps track of the last update time the map information sent to each client and sends only the changes from this time on. This information is compiled by traversing the DAG and including only the nodes since the last client update. This partial DAG is sent to the client as a JSON object. If nothing has changed since its last update, a client would receive an empty JSON object.

Fig. 3 shows an example JSON object returned by the state server, and the resulting map objects to be displayed on the

[1]To learn which XMLRPC functions to use for different actions, such as setting the view, the developers can query the state server for a list of available functions and the signatures of these functions.

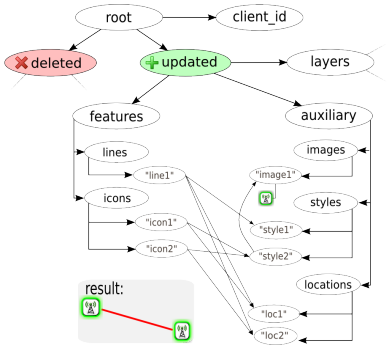


Fig. 3. An example JSON object sent to a client with *client_id*. The end result is node icons, connected with a red line.

client browser (the figure contains only a subset of the edges for the sake of simplicity of illustration). Note that while measurement tools and postprocessing script must be written by the Bowlmap user, from this point on, the user does not need to know how the state server maintains the network state.

The partial DAG sent to the client in a JSON object contains three types of child nodes (see Fig. 3). The *client-id* contains the identifier assigned by the server to the client. At first, the client receives the entire map, but for the next queries uses this identifier to receive only the updates. Under *client-id*, there are two subgraphs: The *deleted* and *updated*. Both *updated* and *deleted*, subgraphs contain the actual elements to be deleted or added/modified (respectively) on the client map such as *layers*, *features*, and *auxiliary* information. The *features* node contains objects to be drawn on the map. The *auxiliary* node contains general and repetitious information such as attributes and styles. For instance, both "icon1" and "icon2" have a "style2" and "image1". Using a DAG representation, we are able to represent the repetitious information such as icon images in a more compact manner and save from bandwidth. Otherwise, for instance, if we had n nodes, we would have sent this image information n times. Note that our solution may increase the processing complexity at the server and client sides, which both need to walk through this DAG to create and process the JSON object, respectively. In Section III-C, we show that indeed our object representation leads to high savings in bandwidth, while the delay for displaying objects remains reasonably low.

III. Performance Evaluation

The effectiveness of visualization is dependent on the application, the data and the user [3]. To evaluate Bowlmap's effectiveness, we evaluate the following [3]:

- Effective summarization and high dimensionality: To monitor and debug the network state with several different components, it is necessary provide an overview of the current situation and visualize several types of information together almost real-time.
- Flexibility: Testbeds undergo several changes. Hence, flexibility is essential to extend the visualization to capture these changes. Furthermore, it is highly desirable that the framework can be configured for entirely different setups.
- Scalability: As a web-based visualization tool, Bowlmap needs to maintain low client response times, which in turn requires low bandwidth usage and short times to process and visualize the map data.

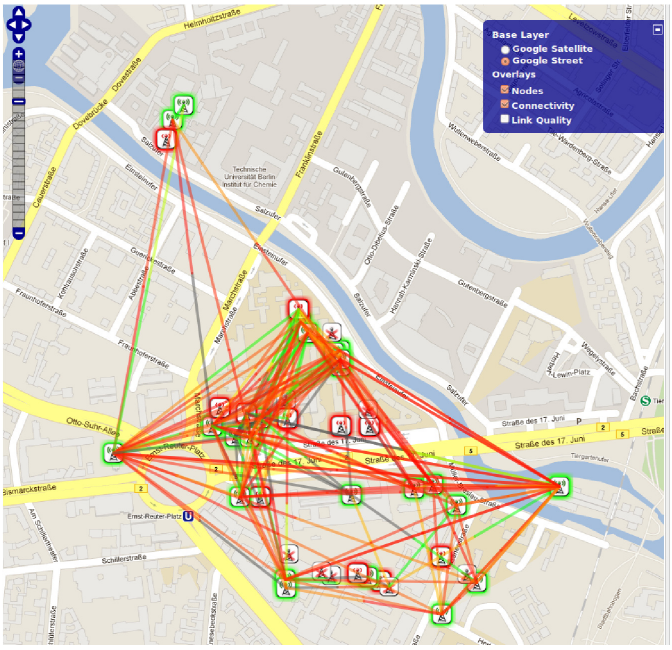


Fig. 4. Bowlmap visualizing the link quality among nodes in the BOWL network. The green nodes are running experimental software, and the red nodes are running our default configuration. The nodes with crossed-over icons are off. The color of the lines represent the connectivity quality: green being the highest quality, and gray represents no connection. The lines are bi-colored, each part representing the connectivity quality in a given direction. Hence, it is possible to view asymmetric links as well as the connectivity quality distribution over all links.

A. Effective Summarization and High Dimensionality

In this section, we show examples from the BOWL wireless mesh testbed. Fig. 4 shows an example visualization of BOWL based on its connectivity quality among different nodes (measured by accounting of beacon frame receptions). Using Bowlmap the answers to the questions that system administrators daily ask to manage their networks are summarized in one map: How is each node performing and how is the network doing overall? What are the critical nodes and links, removal of which, partitions the network? What faults are present, and where are they located? The effect of fixes are also immediately visible, for instance, when nodes are displayed with their correct icons.

Next, we describe a small example to show how Bowlmap is able to bring together different types of information. The main goal in this example is to monitor the operation of a resource allocation controller in the BOWL network. The resource allocation controller assigns the modulation rate and transmission power on a per link basis. Its main goal is to maintain the throughput performance comparable to the case

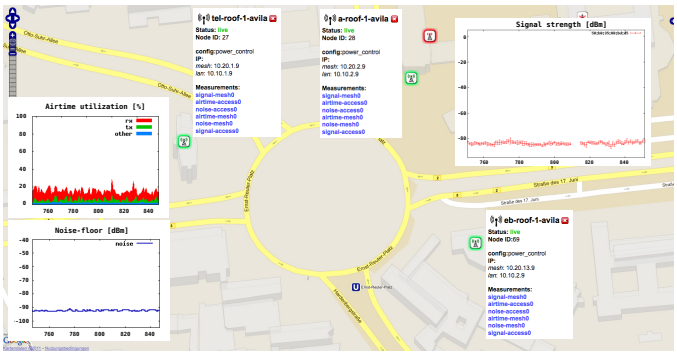
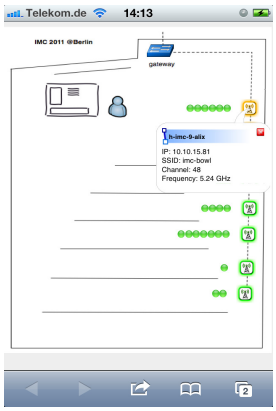
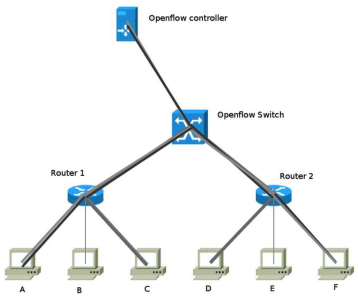


Fig. 5. An example scenario where both PHY layer statistics and the power/rate assignments are visualized to debug a resource control algorithm.



(a) Iphone screenshot showing the number of users connected to BOWL APs in IMC 2011.



(b) A screenshot of Bowlmap visualizing an OpenFlow network.

Fig. 6. Bowlmap used in different scenarios.

with the maximum transmission power. Fig. 5 depicts an example: the sender *eb-roof-1-avila* transmits to two receivers, *tel-roof-1-avila* and *a-roof-1-avila*. The link color corresponds to transmit-power-level on the sender side, where maximum is shown with red, medium power with yellow, and minimum with green. The thickness of the link stands for the modulation rate on the sender side, the thicker the line the higher the chosen modulation rate. In this set-up, we see that the link from *eb-roof-1-avila* to *tel-roof-1-avila* (Link 1) has a red color, and it is less thick compared to the link to *a-roof-1-avila* (Link 2). This confirms that Link 1 requires a higher transmit power and a lower modulation rate compared to Link 2. Furthermore, to monitor the environmental conditions during the experiments, the measurement postprocessing script sends different PHY layer information (like the noise level, the airtime utilization of the radio interfaces and received signal strength indicator (RSSI)) as constantly updated time series graphs to the state server (see Fig. 5). Without Bowlmap, this experiment can only be performed by opening several terminal windows displaying output of several different measurement scripts for noise, airtime utilization, and RSSI, and *tcpdump* for each node, which becomes significantly hard to follow as the number of nodes increases.

B. Flexibility

In this section, we give 3 examples to show the flexibility of Bowlmap.

1) Visualizing a new wireless set-up: We used the Bowlmap to monitor user associations to the newly deployed BOWL access points (APs) during the IMC 2011 conference, for which the BOWL team provided the Internet access. Thanks to the browser-based operation, we could display the map in any web-enabled phone. Fig. 6(a) shows a screenshot taken from an Iphone, which shows how many clients (the green dots in the figure) are attached to different APs (the AP icons). Furthermore, by clicking on the AP icons, we get additional information including the IP address, ssid of the AP as well as the channel and the frequency the AP is operating.

2) Visualizing a mobile network trace: We used the traces from multi-hop mobile wireless network in the MANIAC challenge [2] in the CRAWDAD website [4] (see Fig. 7). In this example, the measurement post-processing script uses a snapshot of the network topology from the topology traces collected between 25-11-2007 and 26-11-2007, and computes the closeness centrality of each node to understand the nodes that play an important role in terms of routing. Essentially, closeness centrality indicates how many adjacent routes a node has, how well it is interconnected with other nodes in the network. We noticed that the topology included disconnected nodes, and computed the modified closeness centrality as $C(v) = \sum_{w \in V - \{v\}} \frac{1}{d(v,w)}$, where d is the distance between nodes v and w, calculated using Dijkstra's shortest path algorithms [5]. In the figure, the node with the maximum closeness value is placed in the center, and the other nodes are randomly placed around this node. The distance to the center node is calculated based on the normalized closeness value (with respect to the node with maximum closeness): the smaller the normalized closeness, the farther away the node is from the center. The radius of the circles around each wireless node as well as its grey-scale value also visually show the normalized closeness value. Clicking on nodes displays additional information about absolute as well as normalized values. Hence, through Bowlmap, we could understand how the nodes are connected to each other in the network (even if we did not have node location information).

3) Visualizing an Openflow network from scratch: The final example is the OpenFlow study by the OpenFlow project team in T-Labs. The team set up their own Bowlmap (with its own state server) in a couple of hours to validate the operation of an OpenFlow switch. In an Openflow network, the controller decides how to forward each packet of a flow, if the Openflow switch does not already have a rule for this flow. In their study, the network was created in Mininet [6] and using Open vSwitch [7]. For their case, the flows are depicted by connecting the nodes that report to be carrying

[2]http://www.maniacchallenge.org

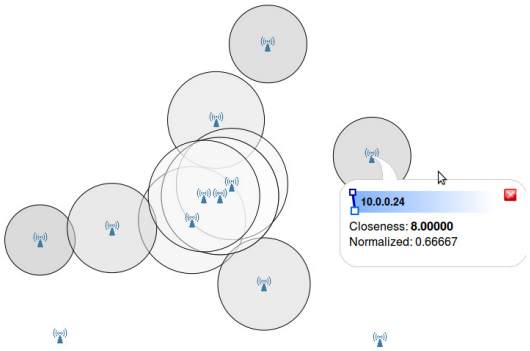


Fig. 7. Closeness centrality of wireless nodes in MANIAC challenge.

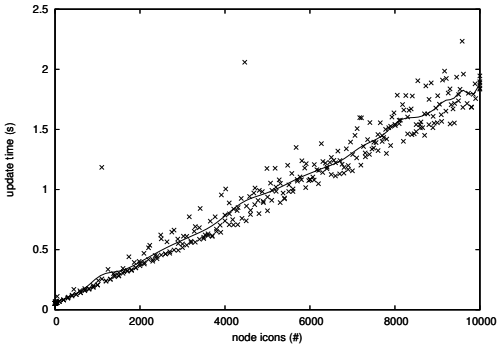


Fig. 8. The update time as the number of nodes increases. The results show that the update time increases linearly and is around 2 s for 10,000 nodes.

the same flow by a line with the same color. The thickness of the line indicates the bandwidth of the flow. Fig. 6(b) shows an example topology with two routers, one OpenFlow switch, one OpenFlow controller, and several hosts. Using the Bowlmap, it was validated that the OpenFlow switch indeed forwards the packets of flow B-E directly, as the rule for how to handle this flow is already installed. For the other two flows with no rules, the packets are first forwarded to the controller. The least-bandwidth consuming flow was B-E and hence, its corresponding line is the thinnest.

C. Scalability

In this section, we evaluate scalability in terms of:

- Network size: How long do map updates take when we have 10,000 nodes?
- Network dynamics: How many map updates are required to represent the BOWL network on average?
- Time: How do the update sizes change when we visualizing clients in a long-term measurement study [8]?

To show the scalability in terms on network size, we artificially generated a varying number of new nodes at each 3 s interval for a total time of $\approx$ 10 mins. We started by adding first 10 nodes until we reached 100 nodes, then started adding 100 nodes until we reached a 1000 and so on. At each update, a full update was sent, which included the "name", "location", "position", and "image identifier" of all nodes. The time of retrieving an update includes all delays that occur from

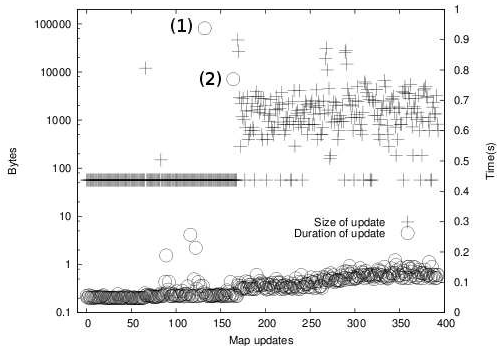


Fig. 9. The size and duration of incremental updates at the client. At (1), the first node updates are received, which is $\approx$ 10 K and requires $\approx$ 0.8 s to be shown on the map. At (2), the OLSR updates start being received, and the size of the updates increases to a few KB s with a display time of $\approx$ 200 ms.

the time the update is requested to the time the objects can be visualized in the browser. These include: generating the update tree and the JSON object, querying the CGI script (which is typically around 50 ms) and HTTP GET Request and Response for retrieving the JSON object. There was no network delay, as we performed this study on the local host. The Fig. 8 shows that the update time indeed increases linearly as the number of nodes increases and reaches to $\approx$ 2 s for 10,000 nodes. Even though this delay could be noticeable by the client, it still is a worst case delay, since, in a typical network, node states and locations do not change significantly to result in full updates.

To understand the scalability in terms of network dynamics, we used the client side logs from the BOWL network, where OLSR was running as the default routing protocol and the post-processing measurement scripts visualized the OLSR topology, which changes continuously. Fig. 9 shows the size of the updates and how much time each update takes. The first non-empty update is received at the 68th update when nodes are added on the map (marked as (1) in Fig. 9). This is a 10 K update, which takes around 0.8 s. Starting from 171th update, the OLSR connectivity information is added to the map (marked as (2) in Fig. 9). From this point on, the size of the incremental updates rise to a few KBs and the duration of updates also slightly increases but stays always less than 200 ms with a few outliers. These results show that Bowlmap maintains an acceptable update time [9], [10].

Finally, to understand the scalability in time, we use traces from Dartmouth campus measurements, which span approximately September 2005-October 2006. For this case, we estimate the overhead of displaying clients associations to the APs deployed using the information available in the syslog traces [8]. We assumed a minimal client object, which is represented by an icon and a line that connects to an AP icon. By choosing minimum-sized icon identifiers and names, we limit the size of this object to 300 B. When no updates happen on the map, an empty object of 144 B is sent back to the requester. We evaluate the CDF of the update sizes at 2 s intervals when incremental or full updates are used. Fig. 10

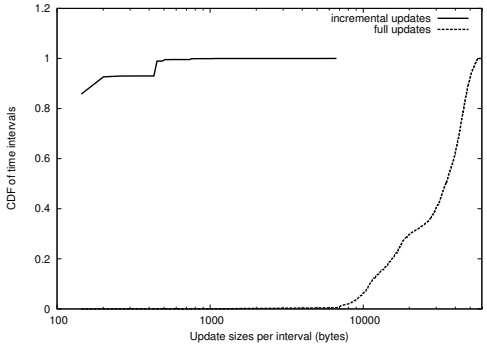


Fig. 10. The CDF of the update sizes for incremental and full updates of visualizing clients associated with APs deployed in the Dartmouth campus between September 2005-October 2006 [8]. We assumed a minimal client object of size 300 B, where an empty update is 144 B.

shows, since the full updates always receive the full list of client icons, 50% of the updates are larger than 30 KB, with the maximum update size of 60 KB. However, 85% of the time there is no change in the Dartmouth network, which results in mostly empty updates for the incremental update case. For the rest of the time, the incremental update size has a long tail from 200 B to 6700 B. These results show that incremental updates successfully maintain low bandwidth overhead, which is essential for scalability of the Bowlmap.

IV. Related Work

The importance of visualization has already been shown for operating systems, where, for instance, visual aids of CPU usage or memory consumption improve configuration management and debugging [11]. A similar emphasis is presented for network visualizations in [12], which proposes a scatter and phase plot animation tool, SPLAT, for mining and visualizing internet measurements, which have both temporal and spatial component.

In the wireless domain, visualization is mainly used to represent the wireless signal quality. For instance, Wiviz [13] monitors a wireless environment, scanning different channels and creates a self-organizing map of nodes in the vicinity. Wireless Network Visualization project [14] merges a point data file produced by wardriving using the Netstumbler [15] software and a high-resolution black and white aerial photography of the same area. The closest to our work is SCUBA [16], which proposes a "focus and context" visualization framework for diagnosing the mesh networks. In contrast to our web-based implementation, the SCUBA visualization engine is a standalone Java application. The application is able to visualize mesh networks from a route, a link and a client context, with the goal of pinpointing problems with more in-depth information using these context switches.

In this paper, we focused on a framework that enables live visualization of network data. Our main aim was to create a visualization framework, that caters to different user needs and is modular enough to support different types of network data representations. While our case studies mainly use network graphs, we do not exclude the possibility of using different graph visualization methods [17], [18], [19]. Essentially, the measurement post-processing scripts can communicate any type of visualization to the state server, which in turn creates the object to be displayed on the client side.

V. Conclusions and Future Work

In this paper, we presented Bowlmap as a flexible tool to visualize measurements with low communication overhead. We showed that Bowlmap can be used in many different scenarios to provide intuitive monitoring and debugging. For future work, we plan to support animations and add the functionality to replay live measurements. Furthermore, we aim to provide a more interactive interface, which allows triggering measurements through the map interface (e.g., starting a traffic flow by just clicking on a node and dragging the mouse to the destination node on the map).

VI. Acknowledgments

We would like to thank Ruben Merz and Julius Schulz-Zander for their initial work on the Bowlmap and, Nadi Sarrar and Robert Wuttke for providing us with screenshots for the Openflow case study.

References

[1] R. Merz, H. Schiöberg, and C. Sengul, "Design of a configurable wireless network testbed with live traffic," in *TridentCom*, May 2010.
[2] T. Fischer, T. Hühn, R. Kuck, R. Merz, J. Schulz-Zander, and C. Sengul, "Experiences with bowl:managing an outdoor wifi network (or how to keep both internet users and researchers happy?)," in *Usenix LISA*, 2011.
[3] N. Marrero, "Visualization metrics: An overview," in *Visualization*, 2007.
[4] A. Hilal, J. N. Chattha, V. Srivastava, M. S. Thompson, A. B. MacKenzie, L. A. DaSilva, and P. Saraswati, "CRAWDAD trace vt/maniac/2007/topology (v. 2008-11-01)," Downloaded from http://crawdad.cs.dartmouth.edu/vt/maniac/2007/topology, Nov. 2008.
[5] V. Latora and M. Marchiori, "Efficient behavior of small-world networks," *Physical Review Letters*, vol. 87, no. 19, 2001.
[6] "Mininet: rapid prototyping for software defined networks," http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.
[7] "The open virtual switch," http://openvswitch.org/.
[8] T. Henderson, D. Kotz, and J. Yeo, "CRAWDAD trace set dartmouth/campus/syslog (v. 2009-09-09)," Downloaded from http://crawdad.cs.dartmouth.edu/dartmouth/campus/syslog, Sep. 2009.
[9] J. Nielsen, "Response times: The 3 important limits," http://www.useit.com/papers/responsetime.html, 1993.
[10] http://blog.kissmetrics.com/loading-time/.
[11] D. Hughes, "Using visualization in system and network administration," in *Usenix LISA*, 1996, pp. 59–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=1029824.1029836
[12] J. Sommers, P. Barford, and W. Willinger, "SPLAT: a visualization tool for mining internet measurements," in *PAM*, Mar. 2006.
[13] N. True, "Wiviz," http://devices.natetrue.com/wiviz/, 2005.
[14] K. A. R. S. Program, "Wireless network visualization project," http://www.ittc.ku.edu/wlan, 2002.
[15] http://www.netstumbler.org/.
[16] A. P. Jardosh, P. Suwannatat, T. Höllerer, E. M. Belding, and K. C. Almeroth, "Scuba: focus and context for real-time mesh network health diagnosis," in *Proceedings of the 9th international conference on Passive and active network measurement (PAM'08)*, 2008, pp. 162–171.
[17] F. B. Viégas and J. Donath, "Social network visualization: Can we go beyond the graph?" in *Workshop on Social Networks (CSCW)*, 2004.
[18] J. Heer and D. Boyd, "Vizster: Visualizing online social networks," in *IEEE Information Visualization (InfoVis)*, 2005, pp. 32–39. [Online]. Available: http://vis.stanford.edu/papers/vizster
[19] "Graphviz-graph visualization software," http://www.graphviz.org.