BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CLUSTER DATA VISUALIZATION FOR INFISPECTOR

CLUSTER VIZUALIZACE DAT PRO INFISPECTOR

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **VRATISLAV HAIS**
AUTOR PRÁCE

SUPERVISOR **Mgr. Bc. HANA PLUHÁČKOVÁ,**
VEDOUCÍ PRÁCE

BRNO 2017

Bachelor's Thesis Specification/20046/2016/xhaisv00

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems | Academic year 2016/2017

Bachelor's Thesis Specification

For: **Hais Vratislav**
Branch of study: Information Technology
Title: **Cluster Data Visualization for InfiSpector**
Category: Algorithms and Data Structures

Instructions for project work:

1. Study methods of general data visualization.
2. Find and provide review of existing solutions for data visualization.
3. Design a proposal for InfiSpector specific needs, or reuse any of existing solutions.
4. Implement the data visualization solution directly within InfiSpector project.
5. Simulate real data and check the result with the expected results. Discuss performance aspects.
6. Discuss results and achievements with InfiSpector community and share source code.

Basic references:

- according to the instruction of the supervisor

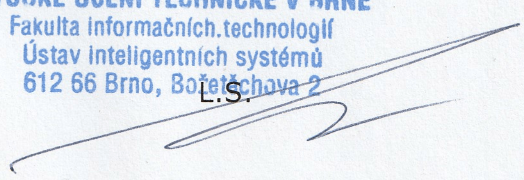
Requirements for the first semester:
Items 1 to 3.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Bachelor's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Pluháčková Hana, Mgr. Bc.**, DITS FIT BUT
Beginning of work: November 1, 2016
Date of delivery: May 17, 2017



L.S.

Petr Hanáček
Associate Professor and Head of Department

Abstract

This bachelor's thesis is focused on InfiSpector data visualization by adding a new graph used as a time selector and by modifying open source diagrams that are designed to fit InfiSpector needs. Theory about general data visualization, description of the InfiSpector tool and libraries used mainly for data visualization are described in the beginning. Next, our customized solution and already existing solutions are also described. Finally, last part evaluate accomplished results and propose possible improvements.

Abstrakt

Tato bakalářská práce je zaměřena na rozšíření projektu InfiSpector a to o přidání nového grafu, který bude sloužit pro výběr časového intervalu a o úpravu open-source grafů pro sledování provozu mezi servery, které jsme zakomponovali do našeho projektu. Teorie o vizualizaci dat je popsána hned ve druhé kapitole a popisuje hlavní cíle vizualizace dat, základní typy grafů a pravidla, jimiž je vhodné se řídit při vytváření grafu. Dále jsou v této kapitole popsány knihovny, které je možné použít pro vytvoření grafu. Projekt InfiSpector je představen v sekci 3.1, kde je popsán hlavní cíl projektu, použité technologie a knihovny, o kterých jsme uvažovali. Následuje popis implemenetace jednotlivých částí, na jejichž vývoji jsem se přímo podílel. Hlavní část této kapitoly tvoří implementace výše zmiňovaných nových grafů. Závěr obsahuje zhodnocení dosažených výsledků, návrh možných zlepšení a plány do budoucna.

Keywords

InfiSpector — Infinispan — Kafka — Druid — Big Data — Network — Cluster

Klíčová slova

InfiSpector — Infinispan — Kafka — Druid — Veledata — Síť — Clustery

Reference

HAIS, Vratislav. *Cluster Data Visualization for InfiSpector*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Pluháčková Hana.

Cluster Data Visualization for InfiSpector

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mgr. Bc. Hana Pluháčková. The supplementary information was provided by Mgr. Tomáš Sýkora. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Vratislav Hais
May 14, 2017

Acknowledgements

I would like to thank Mgr. Bc. Hana Pluháčková for accepting and leading my bachelor's thesis and providing useful tips. Also, special thanks to Mgr. Tomáš Sýkora for all the consultations and help. Without him, this work could never be done.

Contents

1 Introduction **3**

2 Methods Of General Data Visualization **4**
- 2.1 Data Visualization . . . 4
- 2.2 Five Basic Rules . . . 5
- 2.3 Existing tools and libraries . . . 6
  - 2.3.1 D3js . . . 6
  - 2.3.2 Chart.js . . . 7
  - 2.3.3 Raw . . . 8
  - 2.3.4 ZingChart . . . 9
  - 2.3.5 Grafana . . . 9

3 Data Visualization In InfiSpector **10**
- 3.1 About InfiSpector . . . 10
  - 3.1.1 Apache Kafka . . . 11
  - 3.1.2 Druid . . . 12
- 3.2 Considered Libraries and Technologies . . . 13
- 3.3 Used Libraries and Technologies . . . 14
  - 3.3.1 Node.js . . . 14
  - 3.3.2 Npm . . . 14
  - 3.3.3 AngularJS . . . 14
  - 3.3.4 Grunt . . . 15
- 3.4 Existing Solutions . . . 15

4 Implementation **18**
- 4.1 Design . . . 18
- 4.2 Chord Diagram . . . 18
- 4.3 BiPartite . . . 19
- 4.4 Time Line . . . 20
  - 4.4.1 Design implementation . . . 21
  - 4.4.2 Implementation of functionality . . . 22
- 4.5 Controller . . . 26
- 4.6 Connection between front-end and back-end . . . 26
- 4.7 Filters . . . 27
- 4.8 Message browsing . . . 28
- 4.9 Grouping . . . 30
- 4.10 Future plans . . . 33

5 Real data simulation, user stories and results **34**
5.1 Real data demonstration . . . 34
5.2 User Stories . . . 37
5.2.1 Bad Coordinator Node . . . 37
5.2.2 Clogged communication . . . 37
5.2.3 Adding New Node . . . 37
5.2.4 Performance . . . 37
5.3 Discussing results with InfiSpector community . . . 37

6 Conclusion **38**

Bibliography **39**

Appendices **41**

A CD content **42**

B Manual **43**

C Poster **44**

Chapter 1

Introduction

The primary goal of this thesis is an extension of already existing project InfiSpector by adding histogram type of graph and modification of the existing ones.

InfiSpector is mainly students project under the guidance of Mgr. Tomáš Sýkora established in year 2015. InfiSpector is a tool for visualization of network communication between servers. Goal of this tool is to visualize communication between servers, so it would be easier for developers to spot issues.

Time line graph should be able to show total number of messages in the interval of 24 hours. Each bar of histogram will stand for one hour and height of bar will indicate total number of sent messages. User will be able to chose desired interval. Graph will spread chosen interval into lower layer with different units and so on until milliseconds. After choosing desired interval we can display communication. This is achieved by one of the two other graphs (depends on demand). Four graphs of the same type will be shown and user can monitor messages he needs to see.

Graphs are implemented in JavaScript with the use of library named D3js [4]. D3js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

Chapter 2

Methods Of General Data Visualization

This chapter contains basic introduction to data visualization, five basic rules of graph creation and available tools and libraries used for data visualization.

2.1 Data Visualization

The main target of data visualization is to interpret data in well arranged and most informative way. Data may be displayed as dots, lines and bars. Data visualization makes data more accessible, understandable and usable. The most common types of graphs are bar charts, pie charts, line graphs and cartesian graphs.

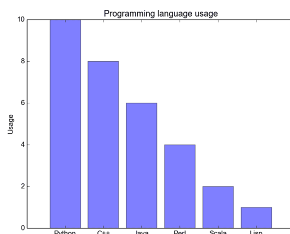


Figure 2.1: Bar Chart [16]

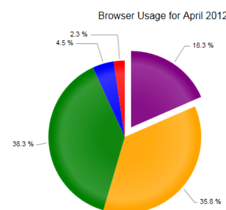


Figure 2.2: Pie Chart [13]

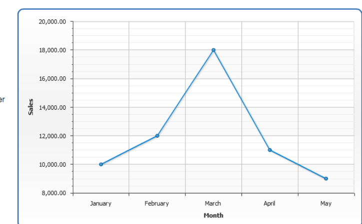


Figure 2.3: Line Graph [2]

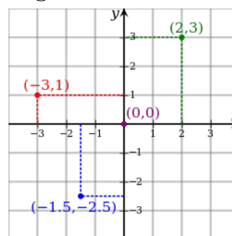


Figure 2.4: Cartesian Graph [19]

Creating a graph is not an easy task. Graphs should not look boring. They need to be both functional and pretty. Before creating graph we have to become conscious about who belongs to our targeted group and what kind of data we want to display. After that we

have to think about how we want to represent this kind of data. This may seem as an easy task, but it's not. Most designers fail to create graph that is pretty and easy to read.

Data visualization takes advantage of a thing called pre-attentive processing [9]. Human can easily tell difference between shape orientation, line length, color. Data visualization takes advantage of this so important values, shapes (etc.) are displayed with some difference so the user can easily spot it. Pre-attentive processing is very fast, because it's done in parallel unlike attentive processing which is done serially. Difference between these two can be seen on Figure 2.5.

24813481187116715541388198443771347915641531845305848641
23475789411484122238814691613548048407890877078678751211
86584234044377134791564153184530584864123475789411484122
23881469161354804840789087707867875121186584234018874276

24813481187116715541388198443771347915641531845305848641
23475789411484122238814691613548048407890877078678751211
86584234044377134791564153184530584864123475789411484122
23881469161354804840789087707867875121186584234018874276

Figure 2.5: Difference between attentive and preatentive processing [9]

2.2 Five Basic Rules

If you want to create perfect diagram you should be following these simple rules:

1. Know your audience
   - It may seem unnecessary, but it's really important. Before you start creating chart you should think about who will be looking at it. You need to know what's important to your targeted group. You need to know if they want one complex, complicated graph with every information or single graph for each information.
2. Use the right type of graph
   - Type of graph depends on type of data you have. For example if you want to illustrate changes over time you should not use pie chart, but line graph. In general for changes over time you should use line graph while categorical data should be displayed on a pie chart or a bar charts.
3. Label axes
   - This is really important. Without labeled axes or any explanation graph is just a decoration. Labeling is necessary so the reader know what scale points are plotted on. Also, you should always start your axis with value 0.
4. Don't use 3D graphs
   - 3D graphs are pretty to look at, but in most cases it's really hard to learn something from them. See for yourself in Figure 2.6

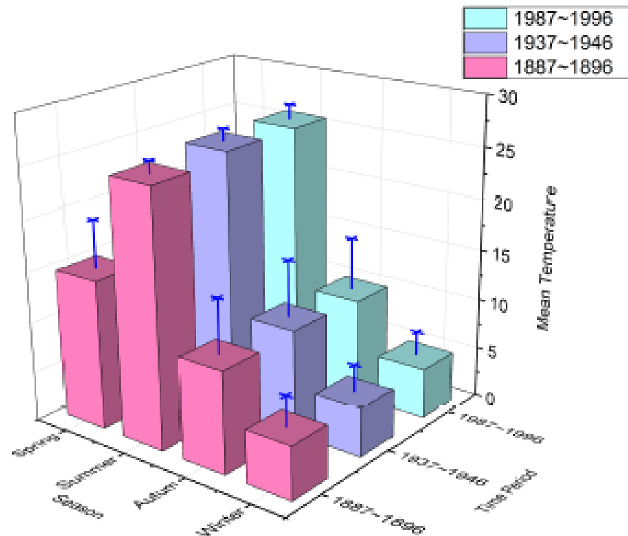


Figure 2.6: 3D bar chart [12]

5. Keep it simple!

   - Animations are pretty (most of the time), but too much animations can lead to a confusion. Same goes with color. Graphs should be simple so the user can immediately understand the purpose of the graph.

2.3 Existing tools and libraries

Some of the existing tools and libraries will be introduced in this section.

2.3.1 D3js

D3js is a JavaScript library for producing dynamic, interactive data visualizations in web browsers. This library is capable of creating really advanced visualizations. D3js offers lots of functions which ease your work with graph creation. There are some basic layouts like histogram (Figure 2.7), tree (Figure 2.8), pie chart (Figure 2.9) and cluster bubble pack (Figure 2.10).

Results may seem difficult and that it took quite a bit of work, but it didn't. Each diagram from Figure 2.7 to Figure 2.10 took about 30 lines of code.

But layouts are not the only benefits you get from importing D3js library. You can also use predefined functions to create axises, select and modify DOM elements and also create nice and swift animations using transitions. In my opinion using transition is much easier in D3js than in CSS.

And what about disadvantages? For a large number of entries manipulation with DOM elements can be extremely slow. Fortunately, we rarely need a great number of entries for a good data visualization.

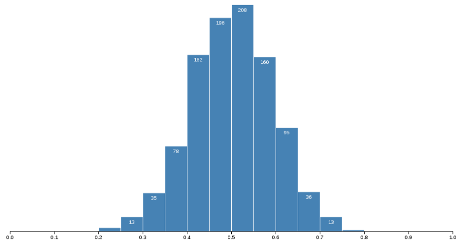


Figure 2.7: Histogram created by histogram layout in D3js

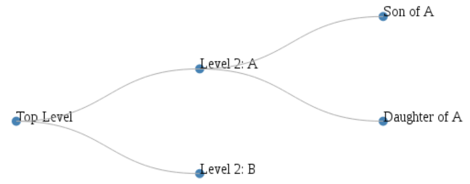


Figure 2.8: Tree layout created by D3js

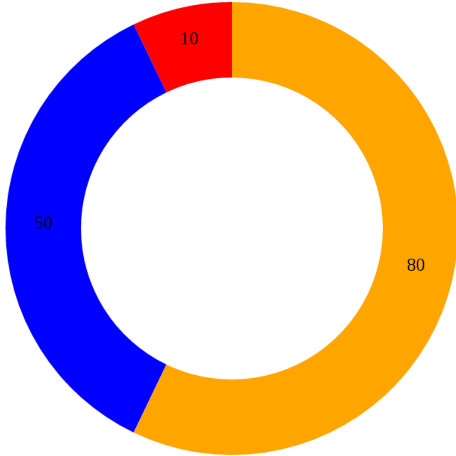


Figure 2.9: Pie chart created by pie layout in D3js

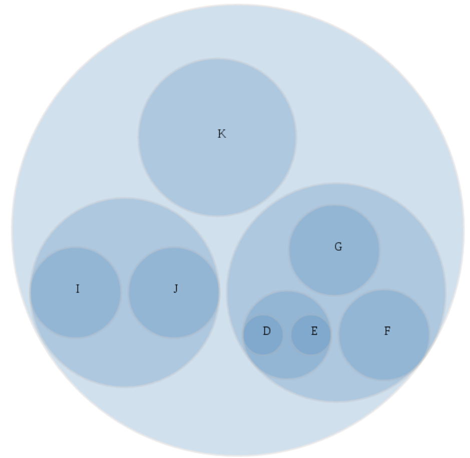


Figure 2.10: Cluster bubble pack created by D3js

2.3.2 Chart.js

Chart.js is one of the most popular open source, charting library for JavaScript. Charts are rendered by using HTML5 canvas elements.

Creating chart is pretty simple. All you have to do is create canvas, select it and put new object named Chart with values like type of graph (bar, line, horizontalBar, etc.), data and colors. And that's it. You can also add some options like axises, borders, etc.

Chart.js is just a charting library. This means the only thing you can do with it is create a simple chart. This may be an advantage, because it's a lot simpler than in D3js, but you can't create anything more advanced like our BiPartite chart (Figure 3.5).

Here are two examples created in Chart.js library:

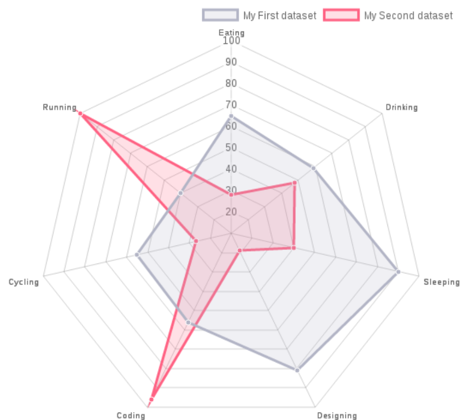


Figure 2.11: Radar chart created with use of Chart.js [5]

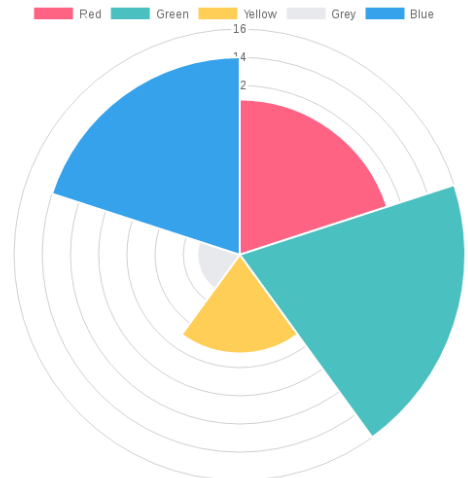


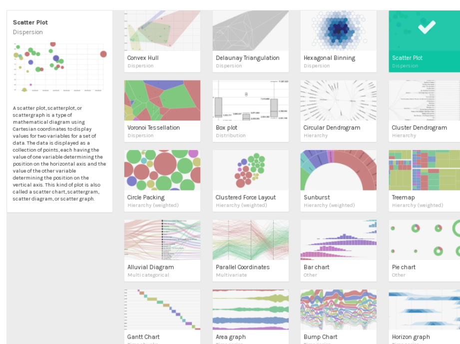
Figure 2.12: Polar area chart created with use of Chart.js [5]

2.3.3 Raw

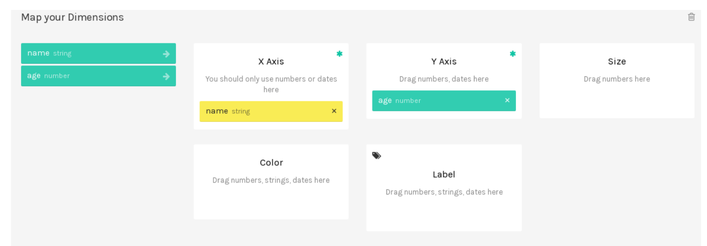
RAW is an open source web tool developed at the DensityDesign Research Lab [7]. Raw work on top of already mentioned D3js (Section 2.3.1) library. Raw allows you to create custom vector-based visualizations and possibility to export your result to png or svg format so you could use it at your web page.

Raw is also highly customizable and extensible. It is possible to create customized chart defined by user. How it works:

1. copy and paste your data into Raw,
2. choose a layout and map the dimensions (Figure 2.13),
3. customize the visualization (Figure 2.14),
4. export the visualization.



(a) Selecting type of graph



(b) Mapping dimensions

Figure 2.13: Choosing layout and mapping dimensions

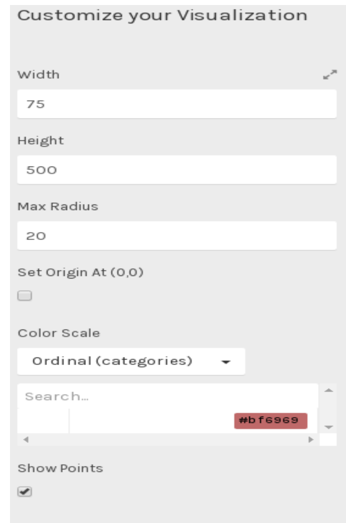


Figure 2.14: Customizing options

2.3.4 ZingChart

ZingChart's huge advantage is his large set of API so you can create interactive charts [1]. ZingChart offers over 100 type of charts that can be used. With CSS-like styling you can creatively customize your charts and design your own themes. Results can be easily exported into various file types like jpg, png and pdf.

Unlike others already described libraries, ZingChart is not free.

2.3.5 Grafana

Grafana is an open source metric analytic and visualization suit which provides powerful and elegant way to create, explore and share dashboards and data [8]. Grafana mainly serve for visualizing time series data for internet infrastructure.

There is a possibility to deploy dashboards on their servers. For one user it is free of charge for up to 5 dashboards. You can also run it on your local machine which is also free.

Grafana is characterized with rich query composing, fast rendering, logarithmic scales, drag and drop panels to rearrange dashboard, quickly adding and editing functions and parameter, etc.

Figure 2.15: Grafana dashboard

Chapter 3

Data Visualization In InfiSpector

The main goal of this section is to introduce project InfiSpector, considered libraries, chosen libraries and existing solutions used in InfiSpector.

3.1 About InfiSpector

The primary target of the project InfiSpector is to create a tool which monitors communication in Infinispan cluster.

Infinispan is an open source project mainly written in Java which is extremely scalable, highly available key/values data store and data grid platform [14]. Purpose of Inifinispan is to make the most of multi-processor and multi-core architectures. It is usually used as a distributed cache, but also as a NoSQL key/value store or object database.

InfiSpector is using technologies Druid and Apache Kafka. The whole architecture scheme and how it works is displayed in Figure 3.1 and described lower in sections.

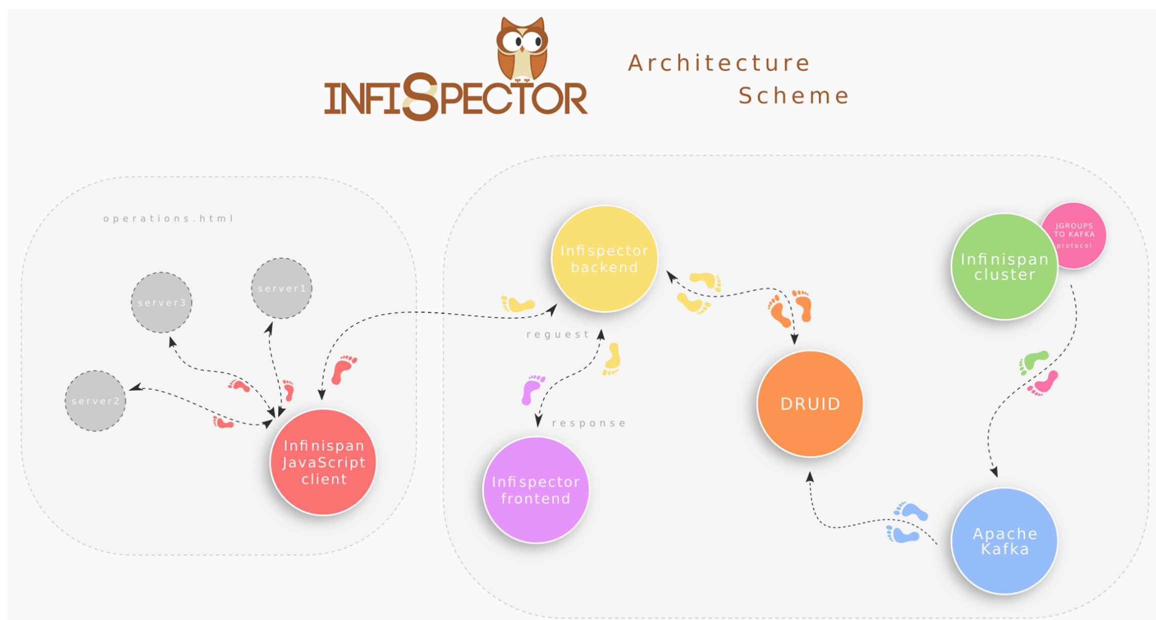


Figure 3.1: InfiSpector architecture scheme

3.1.1 Apache Kafka

Apache Kafka is an open source project developed by the Apache Software Foundation [3]. Kafka is written is Scala and Java and is good for:

- building real-time streaming data pipelines that reliably get data between systems or applications,
- building real-time streaming applications that transform or react to the streams of data.

Kafka is run as a cluster on one or more servers and is used as a storage for streams of records in categories called topics. Each of these records are composed by key, value and timestamp.

There are four core APIs in Kafka:

- Producer API,
- Consumer API,
- Streams API,
- Connector API.

The description of the APIs from the Apache Kafka documentation:
„*The Producer API allows an application to publish a stream of records to one or more Kafka topics.*

The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.

The Streams API allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.“ [3].

The communication between clients and servers is provided with a TCP protocol. Figure 3.2 shows graphically the connection between these four APIs.

In InfiSpector, Kafka is used to capture Infinispan communication and send it to Druid.

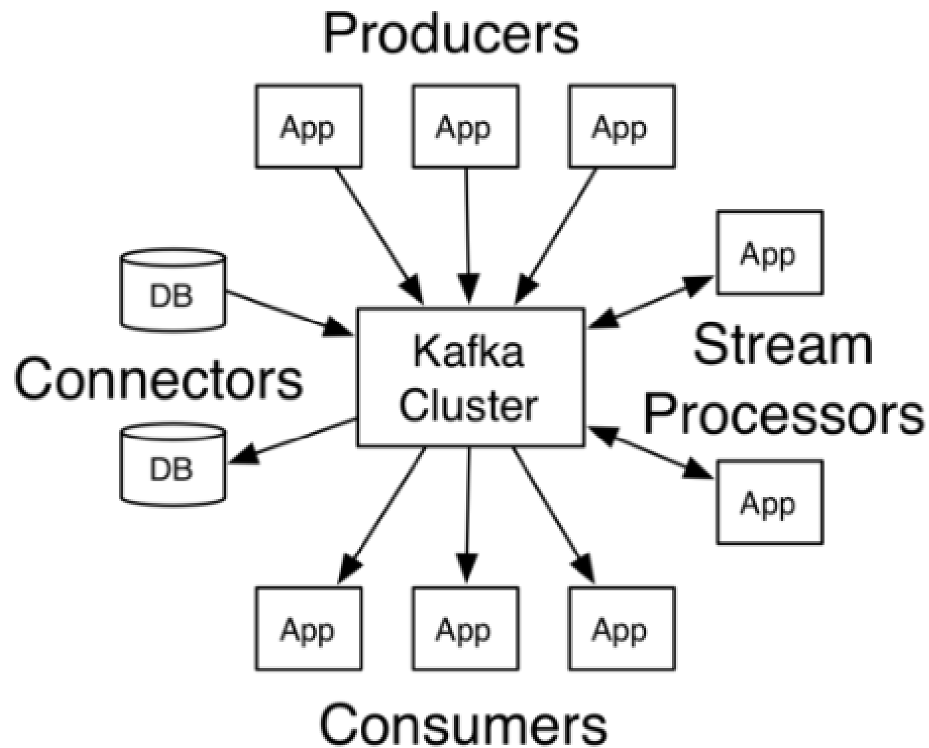


Figure 3.2: Kafka's core APIs connection [3]

3.1.2 Druid

Druid is an open source, column-oriented data store written in Java designed for online analytical processing queries on event data [11]. Druid is designed to quickly absorb massive quantities of event data and provide low latency queries on top of the data. Key features:

- Druid is able to aggregate and filter data in milliseconds.
- Really low latency between when event happens and when is displayed. Latency is caused only by the time that it takes to deliver new event to druid.
- Druid can be used by thousands of concurrent users.
- Cost effective.
- Druid supports rolling updates so you are able to browse your data and use query even during software updates.
- Every second Druid can handle trillions of events, thousands of queries and petabytes of data.

As said earlier, druid is column-oriented which means that each column is stored separately. Only columns somehow related to a query are used in that query. Different columns can also have different indexes associated with them.

Druid is using JSON as a quering language. Although the community has contributed numerous query libraries in a lot of languages. Druid is currently not supporting join operations.

A Druid Cluster is composed of several different types of nodes:

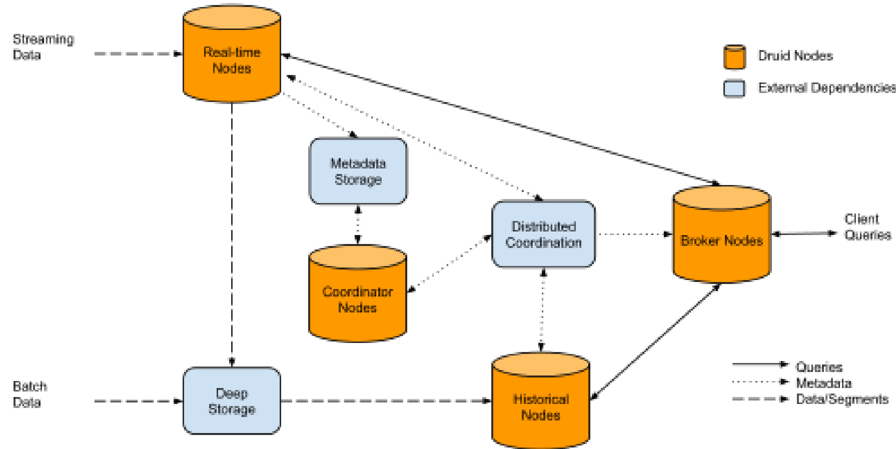


Figure 3.3: architecture of the cluster, including both Druid nodes and external dependencies [20]

- Historical Nodes – download immutable segments locally and serve queries over those segments.
- Broker Nodes – responsible for scattering queries and gathering and merging results.
- Coordinator Nodes – tell historical nodes to load new segments, drop old segments, and move segments to load balance.
- Real-time Processing – involves ingesting data, indexing the data (creating segments), and handing segments off to historical nodes. Data is queryable as soon as it is ingested by the realtime processing logic.

There are few dependencies, which Druid requieres:

- Zookeeper for intra-cluster communication.
- Metadata Storage to store metadata about segments and configuration.
- Deep Storage acts as a permanent backup of segments.

3.2 Considered Libraries and Technologies

During the first suggestions we were considering usage of a new Angular2.0. Reason why we refused it is simple. It was very new, so there were no guides and it was still in development.

Gulp or Grunt? This was also a question during the first suggestions. We have made a simple research about which one is better, but Gulp had same problem as Angular2.0. It was relatively new and the community was not as large as for Grunt.

Also we were considering which library is the best for data visualization. We have found Grafana. We have not used this library because it is specialized mostly on dashboards and there is no possibility to create highly customizable diagrams as in D3js. Possibility to customize our own diagrams is pretty important because we need to react on Infinispan community comments. Also Grafana does not contain any graph for the visualization of the communication in N-node cluster. Another disadvantage is that it needs to be downloaded before you can use it. This could be problematic for some targeted users.

3.3 Used Libraries and Technologies

For a correct function of InfiSpector a user has to have Druid (described in Section 3.1.2) and Apache Kafka (described in Section 3.1.1) installed on a local machine. InfiSpector is also using npm for installations, AngularJS for dynamic parts of a web page, Grunt as a starter and D3js (Section 2.3.1) as library for visualization.

3.3.1 Node.js

Node.js is a server-side platform developed by Ryan Dahl in 2009. Node.js is used for easily building fast and scalable network application. It uses an event-driver, non-blocking I/O model that makes it efficient and lightweight. Node.js also provides a rich library of various JavaScript modules.

Some important features:

- asynchronous and Event Driven,
- very fast,
- single Threaded but Highly Scalable,
- no buffering,
- MIT license[1].

Because Node.js is an asynchronous server he never waits for an API to return data, he simply continues with the next API[18].

Single threading, non-blocking I/O calls allows Node.js support tens of thousands of concurrent connections. To accommodate the single-threaded event loop Node.js is using the *libuv* library.

3.3.2 Npm

Npm is used by JavaScript developers and is part of the *Node.js*. It makes code sharing a lot easier, because you can install every dependency and new updates with just one command in command line [15]. Another advantage is that your team can use package done by people who were already focused on the similar problem and your team may just continue and extend this piece of code. Over 280 000 packages are available for installation with npm. They can be found on the npm website. Disadvantage is, that not every package is secure. Some may be insecure or even malicious.

3.3.3 AngularJS

AngularJS was founded by Google in 2009 [10]. It is a web framework created in JavaScript, that allows to create a single-page application. Application is still programmed in HTML, but extended with some special formatting symbols. Data binding and dependency injections eliminates much of a code you would have to write in HTML.

[1] https://raw.githubusercontent.com/joyent/node/v0.12.0/LICENSE

3.3.4 Grunt

Grunt is a JavaScript task runner written in *Node.js*. With help of Grunt we are able to change code and see results without any need of compiling or restarting application. All we have to do is just refresh the page and results are visible. Grunt uses command line to run custom commands defined in a file known as Gruntfile. Grunt is also extend-able by plugins. There is more than 5 000 available plugins and also a huge community support.

3.4 Existing Solutions

We were looking for some existing graphs made in D3js that could match our needs for visualization in clusters. We have found two. Chord Diagram (Figure 3.4) and BiPartite (Figure 3.5).

Problem with Chord Diagram (Figure 3.4) is that it is too difficult to read and diagram gets unclear when we have more than 20 nodes. We can add more colors but it's still a special type of pie chart so it's not proper to display high number of nodes. Also it had to be modified so it is capable to work with JavaScript promises and with our type of data. In the end, onclick function was added so the user can click on the desired communication and browse messages.

In the end we have found Chord Diagram too unclear and there is no point in keeping it in InfiSpector. It will be removed in the next version.

BiPartite is a bit more transparent than Chord Diagram and most importantly, does not get confusing with a higher number of nodes. When the node is targeted by cursor, it expands. Expanding is getting slower with higher number of nodes. There are six colors used as a differentiation between nodes (Figure 3.5). This graph was also necessary to modify. Also our Gruntfile is very strict and forbid equation in condition with only two equal signs. As in Chord Diagram, BiPartite needed onclick function to be implemented too.

Figure 3.4: Chord Diagram

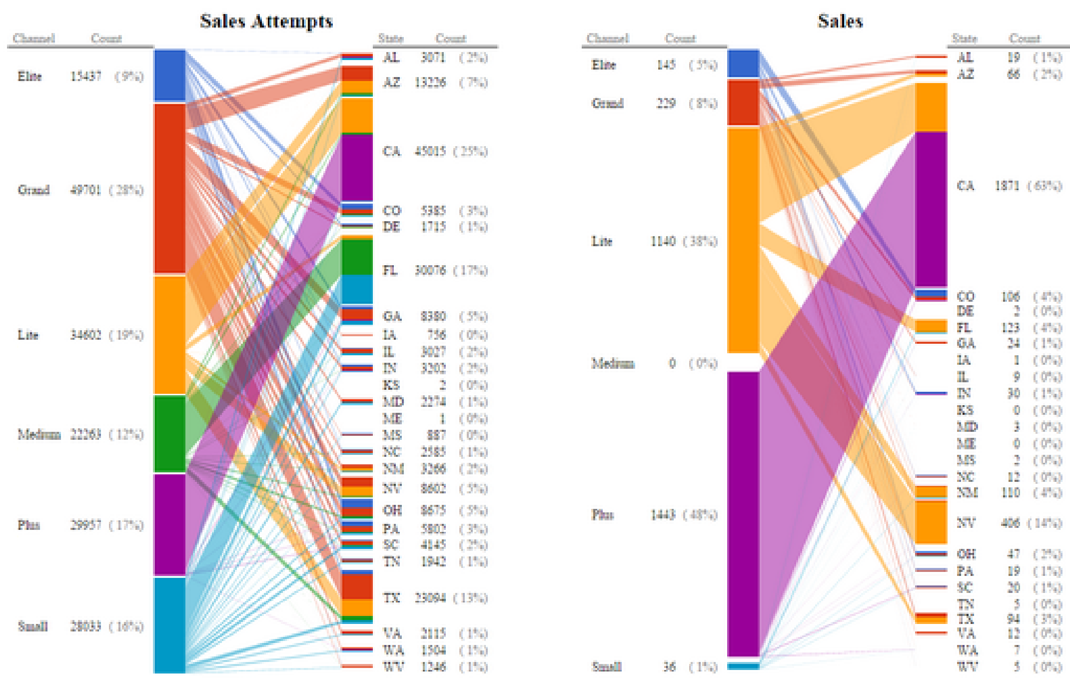


Figure 3.5: BiPartite

Chapter 4

Implementation

This chapter is focused on design and implementation. I will describe in detail each of my contribution to InfiSpector.

4.1 Design

We were discussing a lot about graph design with InfiSpector community. Many ideas were used, many were not. First we had to find some diagrams to visualize message flow. We have found Chord diagram and BiPartite diagram. Recently, we found out that Chord diagram is too unclear and it will be better to remove it and maybe find some more transparent replacement.

There was a lot of discussing about a time line diagram. We were discussing how it should look and what it should display. I have shown some graph templates and we decided that bar chart would be ideal for this purpose.

Design of time line is pretty simple. Bars are blue, targeted are red and selected are green. These colors were chosen randomly and community liked it, so there was no need to change it. X–axis shows time, Y–axis number of messages. Also, number of messages is shown on top of the bar when the bar is targeted.

4.2 Chord Diagram

Chord diagram was our first added diagram in InfiSpector. On Chord we have tried functionality of InfiSpector for the very first time.

We have found Chord Diagram on the official D3js page with examples. The MIT license allowed us to use it. There were some minor problems when trying to compose it to InfiSpector but after removing these conflicts it was working perfectly.

I had to add some functions so it would perfectly fit our needs. One of these functions is on click function. While user clicks on node, function sends request on Druid database to receive all messages sent by this node. These messages are displayed in the message box where user can browse them one by one. Thanks to D3, adding this functionality was pretty simple. All I had to do was add this line of code:

```
1  .on("click", function(d) {clickedNode(gnames[d.index]);});
```

create a new function:

```
function clickedNode(nodeName) {
        angular.element(document.getElementById('ctrl'))
        .scope().getNodeInfo(nodeName);
}
```

When user clicks on a text on click function gets the clicked text and pass it to function `clickedNode`. This function calls a function from our angularJS controller with clicked text as an argument. The function from controller looks like this:

```
$scope.getNodeInfo = function (nodeName) {
    $scope.index = 0;
    var request = $http.post('/getMessagesInfo',
    {
        "nodeName": nodeName
    });
    request.then(function (response) {
        var parsed = JSON.parse(response.data.jsonResponseAsString)[0];
        $scope.nodeMessagesInfo = [];
        for (var i = 0; i < parsed.result.length; i++) {
            $scope.nodeMessagesInfo[i] = "\nnode name: " + nodeName
            + "\ncount: " + parsed.result[i].length + "\nmessage: "
            + parsed.result[i].message + "\n\n\n" + (i + 1) + "/" +
            parsed.result.length;
        }
        $scope.messageInfo = $scope.nodeMessagesInfo[0];
    });
};
```

At the beginning we can see variable `index`. This is a global variable for message browsing. This will be described in Section 4.8. On line 3 we can see declaring a new variable `request`. In this variable we sends command `getMessagesInfo` to Druid with name of clicked node as a parameter. This is done as a JavaScpript promise that is used for asynchronous process of function. When the Druid completes computations and returns results, function continues on line 8. Here we have to process result from Druid and parse each message. Parsed messages are stored in a controller variable `nodeMessagesInfo` and first message is displayed in a message box. More detailed description of message displaying will be in Section 4.8.

As said earlier, we found Chord diagram confusing and unclear. This is why it will be removed from InfiSpector in the next update.

4.3 BiPartite

Adding BiPartite diagram into InfiSpector was harder than adding Chord diagram. The main reason was incompatibility with our *Gruntfile*. Our *Gruntfile* was very strict and forbid different type comparison (unable to use just two equal signs for comparison – must use three), decimal numbers lower than one and greater than minus one must be typed with zero in front of decimal point (e.g. typing .15 is forbidden – you have to type 0.15) and so on. Fixing decimal numbers in code was easy but a real problem were expressions. Solution was to allow it in *Gruntfile*.

After dealing with compiling problems I had to add it to our *index* page, connect it with back–end and some displaying changes.

Adding chart to *index* page was more complicated. I had no clue how to connect data matrix (matrix filled with dummy data for testing purposes) to this chart so I simply added

it as a new script in index page. This was not a good solution because later on we would need matrix received from Druid. So I moved parsing of matrix to BiPartite code and created a new function which appends new `<div>` with BiPartite diagram and then sets everything needed for creating this graph. This solution with some changes persists until now.

Original BiPartite code was displaying two diagrams with one execution and was designed as some sort of a Sales diagram. Also it was not able to colorize more than six nodes and longer name of nodes was also a problem – names were overlapping with numbers. To display only one diagram with every execution was really simple. Author of this diagram was just calling it twice with different names. Deleting it did the trick. With solving this I have also found how to rename diagram. Colorizing was really simple too – there is an array on colors over which it iterates. After 6th node the colors are repeating.

Overlapping was much more bigger problem. We wanted to display more than one diagram on a line so I could not expand it over whole line. Also if I would expand it over whole line and put node names on the edge with diagram in the center, it would certainly not look good. Solution which remains was to find the node with longest name and count the number of characters. This is done with `forEach` function:

```
1 matrix.forEach(function (element) {
2     if (element[2].length > longestCnt) {
3         longestCnt = element[2].length;
4     }
5 });
```

Number of characters is stored in variable `longestCnt` which is passed as an argument to a function creating diagram itself. In this function are calculations acquired by a trial–and–error procedure. I have came up that by multiplying it with 10 and adding 4 will be almost ideal. This number is then added to original number from original author. Unfortunately with extremely long node names (40 and more characters) it starts to overlap again.

Connecting of front–end to back–end will be described in Section 4.6.

4.4 Time Line

Time line diagram is my first designed and programmed diagram from scratch.

We were discussing a lot about functionality of this diagram. Axis were clear. Y–axis for the number of messages and X–axis for a time. The unclear part were on click function and time selections.

First idea was that user will click on every bar of which time he wants to see and than confirm it with a button which would lead to a lower layer with different time units. Problem with this approach is that user must click on every bar and he can skip a bar between time interval which would mean a problem.

Another idea was that user can select only one bar. Implementation of this idea would be simple but it would be ineffective for debugging. It's better to select time interval because problem can be stretched through it.

We have decided that it would be best if user could select only two bars and we will approach to it as an selected time interval.

4.4.1 Design implementation

Transition between blue and red is done by D3js function `transition()` and in code it is done like this:

```
1  thisBar.transition().attr("fill", "red");
```

Variable `thisBar` contains an object of a single bar pointed by the cursor. Upon this object we call function `transtion()` which creates an animation – it slowly changes attribute fill to color red and shows number of messages on top of the bar. When the cursor moves out there is a similar line of code, expect it fills the bar back with a blue color.

Selection of a bar is accomplished by on click function. I will describe it more in a next subsection.

Axis are created with the use of a D3js function named `axis`. Axis took a great amount of time because there was a need to modify their look. The lines were too wide so you could not see ticks on them. In a new version of D3js (version 4.8) it is a lot easier but in a version we are using (version 3) line design must be modified by css. In our case:

```
1  .axis {
2      font-size: 10px;
3  }
4  .axis line, .axis path {
5      fill: none;
6      stroke: #000;
7  }
```

Creation of Y–axis was harder than X–axis because we can have a lot of messages and with higher numbers we need to space them further from axis. Solution for this problem was usage of unit multiplication (kilo, mega, giga). D3 library also has a function for this:

```
1  var y = d3.scale.linear()
2          .domain([highestValue, 0])
3          .range([0, height]);
4  var yAxis = d3.svg.axis()
5              .scale(y)
6              .tickFormat(function (d) {
7                  var prefix = d3.formatPrefix(d);
8                  return prefix.scale(d) + prefix.symbol;
9              })
10             .orient("left");
```

Here we can see setting some variable `y`. This is creation of range for axis with use of d3 function `linear`. Domain is a range of values and range itself tells on how long axis it have to be mapped. For greater understanding see Figure 4.1. This variable is used in axis creation. Axis is created with d3 function `axis` which requires giving it a scale and orientation. The tick format sets displayed values with each tick on axis.

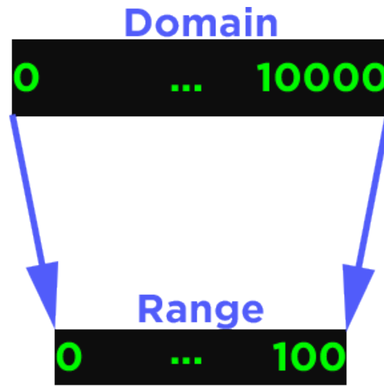


Figure 4.1: Mapping domain into range [6]

4.4.2 Implementation of functionality

Histogram creation

The first problem I was facing during implementation was fact that I could not fully use D3js histogram layout. With this layout values are distributed automatically and the spacing, height and width of bars are calculated automatically. The reason I could not use it is that it distributed our data wrongly. Solution to this was a creation of my own array with conversions and data.

Conversions were discovered by attempt–and–failure method. I came up with a numbers 9.55 for 60 bar diagram and number 24 for 24 bar diagram. This number is used for calculation of space between bars on X–axis.

And this is how the array of values is filled:

```
1 for (var i = 0; i < timeStamps; i++) {
2     histogram[i].dx = width / timeStamps;
3     histogram[i].x = (i + 1) * barWidth - barWidth;
4     histogram[i].y = data[i].numberOfMessages;
5     histogram[i].d = data[i].timeStamp;
6 }
```

For loop iterates 24 or 60 times depending on a time unit. In every iteration I fill array with value `dx` which serves as a width of bar. It is calculated as the width of whole svg element (variable `width`) devided by number of bars.

The variable `x` contains value of the beginning of bar on X–axis. This value is calculated as an order of bar multiplied by width of one bar.

The variable `y` contains number of messages. The value is used for calculation of height of bar.

The last variable `d` is just an auxiliary value. This value have no part in graph shaping and is there just as a storage so I can easily get time value from bar.

On click, mouse over, mouse out functions

Firstly, I will describe mouse over and mouse out functions. Their implementation was pretty simple. The only think these functions are doing is changing color of the targeted bar and adding or removing a number on top of bar. The color won't be changed if the bar is already selected. And this is how function looks like:

```
.on("mouseover", function () {
    var thisBar = d3.select(this);
        thisBar[0][0].nextSibling
        .setAttribute("style", "opacity: 1");
        if (parseInt(thisBar.attr("selected"), 10) === 0) {
            thisBar.transition()
            .attr("fill", "red");
        }
    })
```

The text (number of messages) is present whole time but it is hidden. On line four I'm setting this text visible. After this there is an examination if targeted bar isn't already selected. If so, nothing happens. Otherwise the color of targeted bar is slowly changed to color red. Function `mouseout` is similar. The only change is on line four (opacity is set back to 0) and seven (fill color is set back to steelblue).

The on click function is more interesting. When user clicks on a bar it's value must be stored for later. Also this bar needs to be marked as selected. For this I have a special attribute `selected` which is set to zero by default. After selecting bar I set this attribute to one. The next think is color change. It is done as at mouse over function. If the user clicks on already selected bar the fill color is set back to steelblue.

As I mentioned earlier, when user selects a bar I need to store some values. These values are stored into global variable. During on click function I use two global variables — to store selected time (variable named `selectedTime`) and to check if the graph isn't on the lowest layer already (variable named `lowestLayer`). Variable `lowestLayer` is not changed during on click function. I just use the value to check if there will be some lower layer. This needs to be checked if the user selects second bar on the same layer. Reason why I need to do this will be described later.

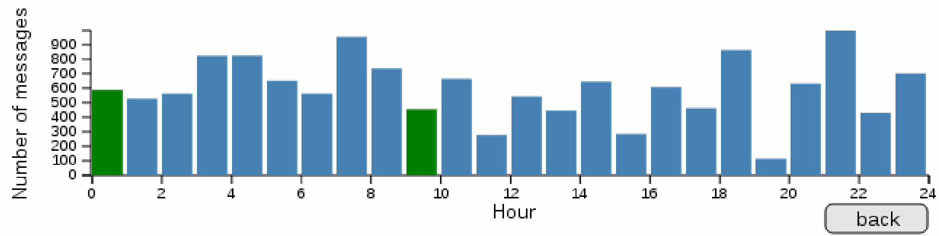
The second variable is changed during on click function. When user selects bar I need to store which time was selected for later. Here comes handy bars attribute `time`. I take this value and append it to already stored values.

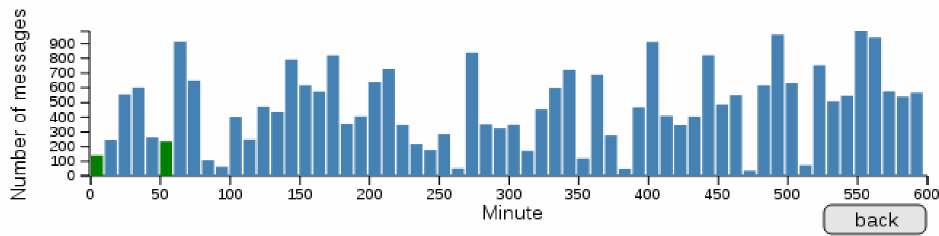
If the user clicks on already selected bar it got deselected. This means I need to remove this time from global variable and set color back to steelblue.

When the second bar on a same layer is selected and graph is not on the lowest layer, the graph will sink into lower layer. The unit of lower layer depends on the interval selection on the higher layer. For better understanding I will demonstrate it on an example:
On the highest layer (hours) the user selects interval from one hour to ten hours. On the lower layer (minutes) one bar represents ten minutes – we have 60 bars and interval of 600 minutes. Now user selects 1st and 6th bar. This means we have an interval of 60 minutes. The function tries if we can display this interval on 60 bars. It can, so the units won't change, but the bar now represents one minute.

User can also return to a higher layer and select different intervals. Switching between layers are implemented with function `clicked`.

While the user decides to return to a higher layer or deselect bar the stored time values in global variable `selectedTime` must be removed. Modifying variable is easier while the layer is changed. In that case the last two values are just removed. With deselection I must first find correct value and then remove it.





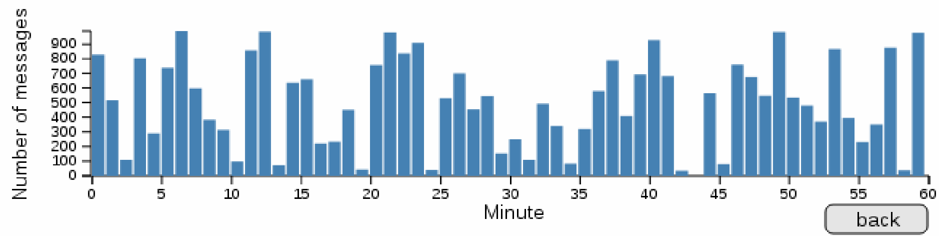


Figure 4.2: Graphical illustration of the example above

Function `clicked`

Function `clicked` is called when two bars are selected. It takes two parameters – actual time scaling (hours, minutes, etc.) and actual multiplier (how much units represents one bar). Within this function two global variables are modified. One of them is already mentioned `lowestLayer`, the other stores individual multipliers which are used to calculate selected time (about this later) and is named `multipliers`.

This function is making decision about the next layer. This is accomplished by an if–else statement. If the multiplier is higher than 60, there is no need in changing units just dividing multiplier by 60 does the trick. If the multiplier is lower than 60, the units are changed. If actual layer was milliseconds and the multiplier is lower than 60, alert is risen and global variable `lowestLayer` is set to one. The alert says: „Unable to go any further!“.

On the end of this function the time line diagram is redrawn with new interval.

Function `higher`

The function `higher` is there for switching to a higher layer. When user clicks on a *back* button in the right corner of diagram this function is executed. All of the global variables are modified during this function because we won't need selected times and multipliers of actual layer and there is no way we stays on the lowest layer (global variable `lowestLayer` is always set to zero). Removing values from global variable `selectedTime` is tricky, because user could already select some bar on actual layer. We need to remove record of actual layer and of a higher layer – user had to select two bars to get to a lower layer so when returning to this layer, we want to start fresh. Normally user can only select one bar before sinking into lower layer. The problem is that on the lowest layer user can select two bars without being switched to a lower layer. With this knowledge I had to came up with a solution considering this factor. In my solution I am taking use of a records of scale units of past layers and actual number of selected intervals in array.

```
1   storage.splice(storage.length - 2 - difference, 2 + difference);
```

In variable `storage` is stored array from global variable `selectedTime`. The function `splice` removes/adds item from/to given array. In our case it removes values from array. The first argument given to `splice` is the starting index from which we wish to remove (add) and the second argument is how much items we wants to remove. The index is calculated as a number of items in array minus 2 (we wish to remove a minumum of 2 items) minus a variable named difference which is basically a number of selected bars on this level.

On the end of this function the time line diagram is redrawn as in the end of the function `clicked`.

Getting selected time from graph

This was the hardest task to accomplish. The reason it was so hard is that user can want to draw diagrams anytime and so I can not rely on him getting to the lowest layer. Another problem is that user can select just one bar (not an interval).

For this purpose I have created function named `getSelectedTime`. This function is using all of the mentioned global variables. As mentioned earlier, user can decide he wants to draw diagram at any point. Because of this there are a lot of if-statements. With every record of new unit in array we need to check if it is not the last time unit. Also we need to take in consideration mentioned fact, that only one bar may be selected on last layer. If only one bar is selected we know for sure that this is the last selected layer.

The function returns an array of two values. First value represents start of the interval in milliseconds and the second value end of the interval in milliseconds.

Function `destroyTimeLine`

Function which purpose is to find occurrence of timeLine on a page and remove it. It is pretty simple:

```
1   function destroyTimeLine() {
2       var element = document.getElementById("timeLine");
3       if (element !== null) {
4           element.remove();
5       }
6   }
```

4.5 Controller

Controllers controll the whole AngularJS application. In our application, controller is used in the dynamic parts of our page and as a mediator between front-end and back-end. We are using two controllers named `InfiSpectorCtrl` and `OperationsCtrl`. I am modifying and working only with `InfiSpectorCtrl` so I will not describe the other one.

In controller there are several important functions that are necessary for our diagrams to work. Among them is a function to get node names and data for our graph. These functions are called `getNodes`, `getFlowChartMatrix` and `getChordDiagramMatrix`. All of them will be datailed described later.

To use controller in HTML it must be added to it. We are adding our controller to our `index.html` like this:
`<div ng-app="InfiSpector" ng-controller="InfiSpectorCtrl" id="ctrl">`
Whole app is enclosed within this `<div>`. This allows us to use AngularJS `ng-click="function()"` and `ng-show="variable"` to call functions from controller or hide/show element depending on a content of a variable in controller.

4.6 Connection between front-end and back-end

Connection between front-end and back-end is implemented within our controller. In controller node names and messages are discovered, matrices with data flow for our diagrams made and functions drawing our diagrams called.

Getting node names

This is the first thing we need to do to continue. We are getting node names from Druid (Section 3.1.2). To call any function in Druid we need to use `$http.post('/functionName')` where `functionName` is a name of a function that you want to be executed. Because this call takes some time and we want to do things asynchronously. This is a reason why we use Promises. A Promise represents a value which may be available now, or in the future, or never [17].

```
1  $scope.getNodes = function () {
2      var request = $http.post('/getNodes');
3      return request.then(function (response) {
4          if (response.data.error === 1) {
5              console.log('ERROR: response.data.error === 1');
6          } else {
7              var nodes = response.data.jsonResponseAsString.replace("[", "").
                   replace("]","").split(",");
8              return nodes;
9          }
10     });
11 };
```

The request to get node names is sent on a line two followed up by a promise. When the response is received from Druid we need to parse these data to get an array of nodes. The result from Druid looks like this:

```
["c03-217a-32519","c03-217a-7681", "c03-217a-76451", "c03-217a-2475"]
```

it is one big string that needs to be edited. This is what is happening on line seven.

We delete every occurrence of bracket and split it into array with a comma as a separator. Array containing node names is then returned.

Getting data for diagrams

Because we are currently using two type of diagrams and each was developed by different author, they require different matrix arrangement. This is why we have two different functions for requesting Druid. They are called `getFlowChartMatrix` and `getChordDiagramMatrix`. The first one return matrix compatible with our BiPartite diagram. The matrix returned from `getFlowChartMatrix` looks like this:

```
1 [["c03-217a-32519","c03-217a-32519",0],["c03-217a-32519","c03-217a
    -7681","2"],["c03-217a-7681","c03-217a-32519",0],["c03-217a-7681","c03-217
    a-7681",0]]
```

First item is name of the node sending message, second item is name of the node receiving message and the last item is the number of messages send.

The matrix returned from `getChordDiagramMatrix` is more minimalistic. It looks like this:
`[[0,2],[0,0]]`
This matrix represents same communication as the one for BiPartite. As you can see, this matrix lacks the name of nodes. This is why the order of items matters. First array represents communication of the first node – first number is number of sent messages to the first node order, the second one is number of messages sent from this node to the second node in order. Order of nodes is set by an array which stores each node name. For three nodes the array could look like this:
`[[1,4,7],[2,5,4],[0,2,3]]`
These functions are called like the function to receive node names with
`$http.post('\functionName')`. The only difference is, that we need to pass argument to this function. There are two arguments – array containing node names and filter (more about filters in Section 4.7). This call is also stored in a variable so we can use JavaScript promise.

```
1 var request = $http.post("/getFlowChartMatrix",
2         {
3             "nodes": nodesArrayInJson,
4             "searchMessageText" : searchMessageText
5         });
```

Lines 3–4 are arguments for function. The function `getChordDiagramMatrix` is called with same arguments.

After receiving response from Druid we can finally draw desired chart by calling its function name and passing array of node names and returned matrix as arguments.

4.7 Filters

Filters create a really big part of InfiSpector. With filters it is easier to monitor communication between nodes. Every session starts with four default filters – *SingleRpcCommand, CacheTopologyControlCommand, StateResponseCommand, StateRequestCommand*. User can easily add their own filter by filling in entry on the top of the page and confirm it with a button next to it. It is possible to add more than one filter at once by separating filters by comma in entry. For each added filter is drawn one diagram.

Clicking a button calls function `addNewGraph` that withdraw content of entry. Content is parsed with regular expression and stored in an array. This array is passed to a function arranging data for specific diagram. This means we have two different function. The decision of which function is called is depending on a user selection (which graph he wants to see).

In those functions is for loop which iterates through this array of new filters and upon every iteration is a request on a Druid with JavaScript Promise. From druid we receive matrix and the process continues as described in a text above.

Figure 4.3: Two different filters. You can see that there is different communication flow

4.8 Message browsing

Sometimes the error can not be spotted in a message flow. In those cases the message browsing comes in handy. You may wonder why it is better to use our tool and not to browse logs without it. The reason is that InfiSpector shows already filtered messages. Before user can browse those messages he is forced to select time interval and draw graphs. If he spots some error in communication flow he can click on a node with that error and browse only those messages that had something to do with that node (he sent or received). We have created special window at our page just for the messages. User browse messages by clicking on buttons. You can see how does it look and even how does a message looks like at Figure 4.4.

Message browsing is implemented within controller and requires two global variables – variable to store all messages (so we won't call druid every time we need next message) and variable to store index of currently shown message.

When user clicks on a node name at diagram it calls controller function `getNodeInfo` with clicked node as an argument. Inside this function we create request on a Druid to receive all the communication containing selected node name in it. When the messages are received they are stored in a global variable `nodeMessagesInfo` and parsed to more transparent version. First message is then assigned to the variable which is shown in the Message List window (Figure 4.4). The function is pretty simple:

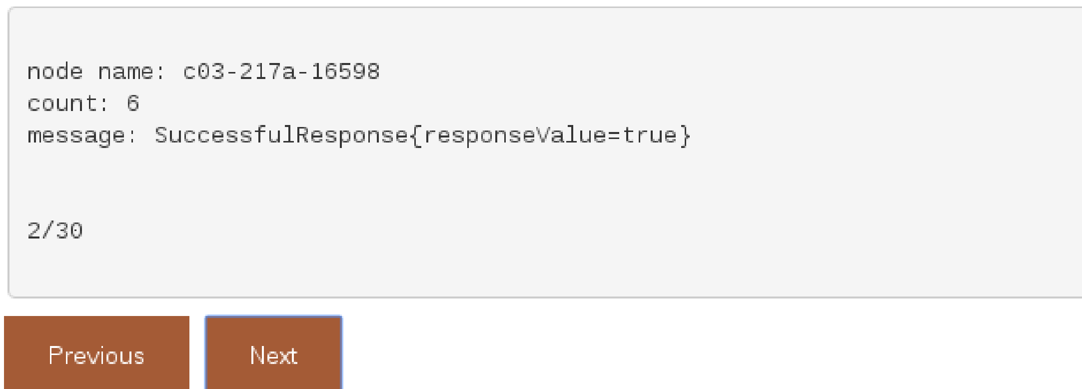


Figure 4.4: Window with second message out of 30

```
1  $scope.getNodeInfo = function (nodeName) {
2      $scope.index = 0;
3      var request = $http.post('/getMessagesInfo',
4      {
5          "nodeName": nodeName
6      });
7      request.then(function (response) {
8          var parsed = JSON.parse(response.data.jsonResponseAsString)[0];
9          $scope.nodeMessagesInfo = [];
10         for (var i = 0; i < parsed.result.length; i++) {
11             $scope.nodeMessagesInfo[i] = "\nnode name: " + nodeName + "\
                   ncount: " + parsed.result[i].length + "\nmessage: " + parsed.
                   result[i].message + "\n\n\n" + (i + 1) + "/" + parsed.result.
                   length;
12         }
13         $scope.messageInfo = $scope.nodeMessagesInfo[0];
14     });
15 };
```

On line 3–6 we can see request on a druid with calling a function named `getMessagesInfo` and passing it argument `nodeName`. This is followed up with a JavaScript Promise. In the body of for loop we can see editing received message to a more transparent version – adding some new lines and spaces.

When messages are stored in a global array and index is in global variable, it is pretty simple to move to a next messages. We have two functions implemented for this purpose. They are called `nextNodeMessageInfo` and `prevNodeMessageInfo`. Neither of them have any argument.

```
1  $scope.nextNodeMessageInfo = function() {
2      $scope.index++;
3      if (($scope.index % $scope.nodeMessagesInfo.length) === 0) $scope.index =
           0;
4      $scope.messageInfo = $scope.nodeMessagesInfo[$scope.index];
5  };
6  $scope.prevNodeMessageInfo = function() {
7      $scope.index--;
8      if ($scope.index < 0) $scope.index = $scope.nodeMessagesInfo.length-1;
```

```
9       $scope.messageInfo = $scope.nodeMessagesInfo[$scope.index];
10  };
```

When user flow over a total number of messages it will display first or last message depending on the button clicked – in case of Previous it switches to last, in case of Next it switches to first. This behaviour is implemented on lines three and eight. As you can see, the global variable `$scope.index` is modified and the message is stored in a variable `$scope.messageInfo` which is displayed in a window.

4.9 Grouping

We have been experiencing performance issues when there were more than 20 nodes. BiPartite diagram wasn't so smooth as normally and Chord diagram was too chaotic. We came up with an idea of node grouping. User is able to set how much nodes forms a group and then press Draw button.

Before grouping starts the value from entry is extracted and stored in a variable. If the value is lesser than 1 notification about error is raised. Otherwise we start with removing `null` node from array because we don't want it to be in a group (in Infinispan's case, `null` node means multicast, i.e. message was sent to every single node in a cluster). After that we loop through the array of nodes assign them to specific group. Groups are stored in a multidimensional array (2D). First dimension is group, second dimension are nodes. This is how assigning looks like:

```
1  for (var index = 0; index < nodes.length; index++) {
2      tmp = Math.floor(index / numberOfNodesInGroup);
3      if (Math.ceil(index / numberOfNodesInGroup) - tmp === 0) {
4          nodesArrayInJson[tmp] = [];
5      }
6      nodesArrayInJson[tmp][index % numberOfNodesInGroup] = {"nodeName": nodes[
           index]};
7  }
```

Variable `tmp` is an auxiliary variable so we are not forced to divide multiple times in one iteration. This variable is determining the group in which node will belong. In `if` condition program checks, if we are going to create a new group. If so, we need to tell JavaScript that on this index will be new array. The last line of code is assigning of a node to it's place in group. We need to add it as a JSON element so it is possible to pass it as `$http.post()`.

Processing and discovering message flow is quite complicated. We have needed four `for` loops to get communication between each group and each node.

```
1  for (var i1 = 0; i1 < numberOfGroups; i1++) {
2      for (var i2 = 0; i2 < groups[i1].length; i2++) {
3          for (var i3 = 0; i3 < numberOfGroups; i3++) {
4              for (var i4 = 0; i4 < groups[i3].length; i4++) {
5                  if (groups[i1].length === 1) {
6                      srcGroup = groups[i1][i2].nodeName;
7                      srcGroup = srcGroup.substr(1);
8                      srcGroup = srcGroup.substr(0, srcGroup.length-1);
9                      console.log(srcGroup + "\n\n\n");
10                 }
11                 else {
12                     srcGroup = "group" + i1.toString();
13                 }
14                 if (groups[i3].length === 1) {
15                     dstGroup = groups[i3][i4].nodeName;
```

```
16                      dstGroup = dstGroup.substr(1);
17                      dstGroup = dstGroup.substr(0, dstGroup.length-1);
18                  }
19                  else {
20                      dstGroup = "group" + i3.toString();
21                  }
22                  promises = promises.concat(getMessagesCountIntern(
23                              JSON.parse(groups[i1][i2].nodeName), JSON.parse(
                                    groups[i3][i4].nodeName),
24                               searchMessageText, from, to, srcGroup, dstGroup))
                                    ;
25              }
26          }
27      }
28  }
```

If the node is alone in a group, the group is named after it. Otherwise the group is named `"groupNUMBER"` where NUMBER is the order of group. On line 22 we have a set of promises which will get us an array of each communicating node. After all promises are resolved we start to create final matrix. We need to connect duplicate record of communication:

```
1   for (var i = 0; i < matrix.length; i++) {
2       for (var j = 0; j < matrix.length; j++) {
3           if (i === j) {
4               continue;
5           }
6           if (matrix[i][0] === matrix[j][0] && matrix[i][1] === matrix[j][1]) {
7               tmp = matrix[j][2];
8               matrix.splice(j, 1);
9               matrix[i][2] = parseInt(tmp) + parseInt(matrix[i][2]);
10          }
11      }
12  }
```

This is why we need to check each item with every other. `If` condition is ensuring that the item won't be compared with itself. If two records on different index are same (`if` statement on line six), we need to add up the number of messages and delete duplicate record. While the process is finished we are able to return the matrix to our controller and pass as an argument to desired diagram to create it.

Under the diagram there is a window which contains a legend. In legend the user can see which nodes are in which group (Figure 4.5)

```
group0:
"c03-217a-49689"
"c03-217a-54471"

group1:
"c03-217a-3385"
"c03-217a-28056"
```

Figure 4.5: Legend to each group

SingleRpcCommand

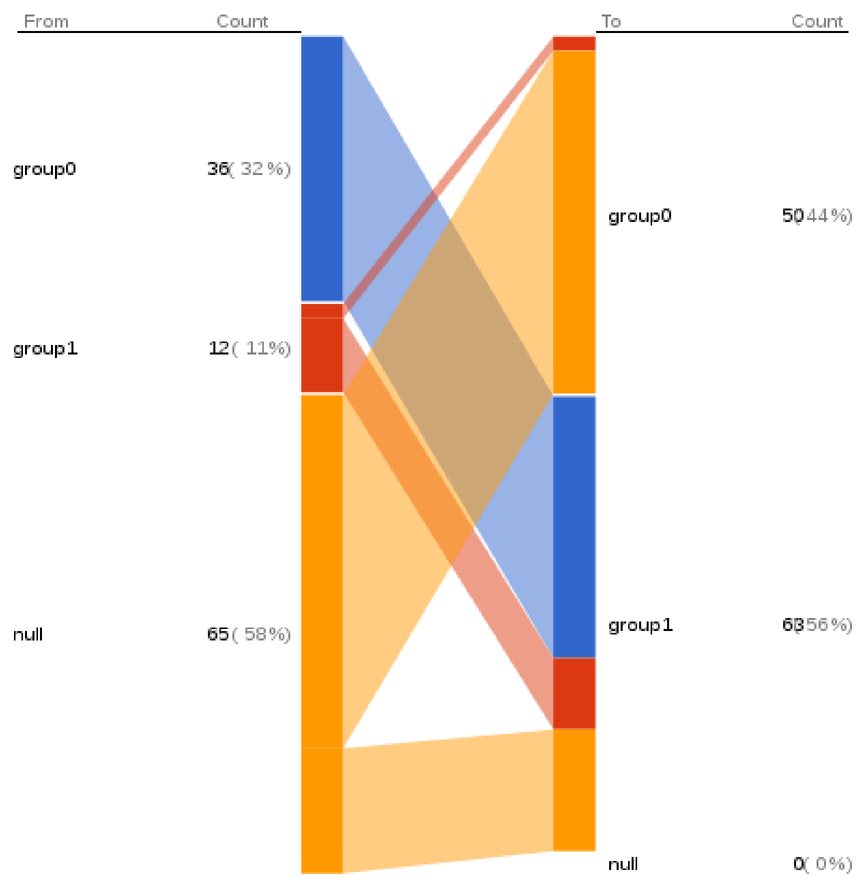


Figure 4.6: BiPartite diagram with grouping corresponding to a legend in Figure 4.5

4.10 Future plans

In the future I would like to modify the code of time line and add new higher layer - days. I will modify all of the current buttons to be powered by SVG so they can be elastic. Grouping at BiPartite diagram is not completely done. I would like to add onclick function which will destroy group and show each node of this group and create animations to this, so it wouldn't be necessary to destroy and redraw diagram. Also we are going to do a performance analysis to see which part slows our tool down and optimize it.

In the next version I will have to adapt on click function on BiPartite, so it would apply filter on message browsing. Progress of our project can be tracked on github[1].

As soon as our tool is done, we would like to cooperate with developers community and add new desired functionality based on community feedback. Also we would like to globalize InfiSpector, so it could be used by any other relevant project community, not only Infinispan.

[1] https://github.com/infinispan/infispector

Chapter 5

Real data simulation, user stories and results

This chapter is about demonstration of InfiSpector visualization of real data flow. Also there will be some made-up user stories which could possibly happen and how InfiSpector is helping to solve it. Last section is about discussion of results with InfiSpector community and link to github with source code.

5.1 Real data demonstration

For demonstration we have a command line command which will simulate node communication. To execute this command we are using `Maven`[1]. Command is `mvn exec:exec` which must be executed multiple times (once for one node) in multiple command lines.

To demonstrate functionality I have created four node communication. Node names are created as a name of my computer (c03-217a), dash and some random number. Results can be seen on figures 5.1–5.3.

[1]https://maven.apache.org/

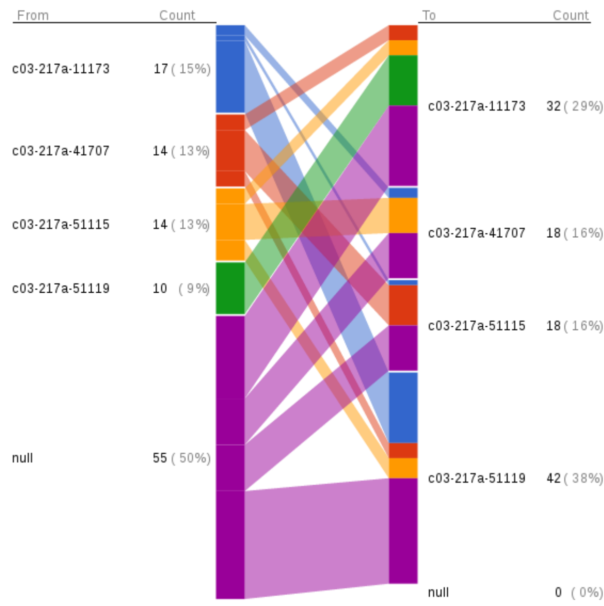


Figure 5.1: BiPartite diagram with real data communication flow

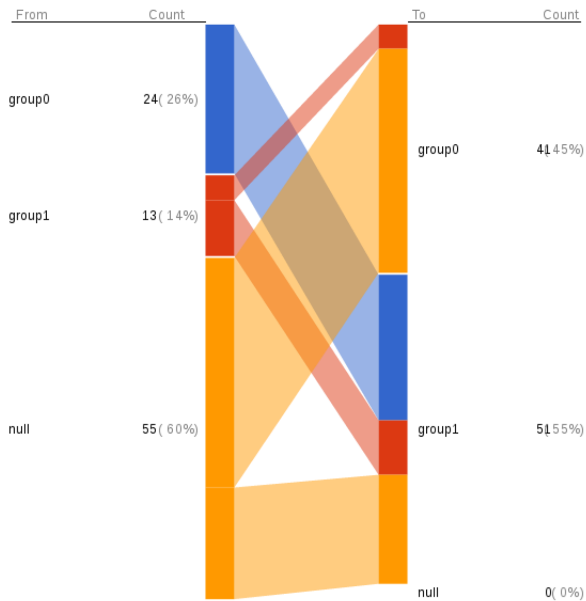


Figure 5.2: BiPartite diagram with grouped nodes

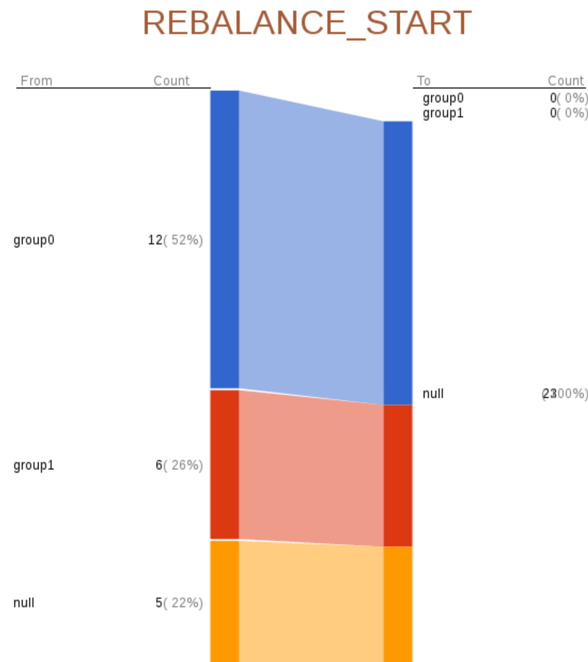


Figure 5.3: Diagram showing communication flow with new manually added filter

Message List

```
node name: c03-217a-51119
count: 4
message: CacheTopologyControlCommand{cache=___defaultcache, type=CH_UPDATE, sender=c0
3-217a-51119, joinInfo=null, topologyId=2, rebalanceId=1, currentCH=DefaultConsistent
Hash{ns = 60, owners = (2)[c03-217a-51119: 30+30, c03-217a-11173: 30+30]}, pendingCH=
null, availabilityMode=AVAILABLE, throwable=null, viewId=3}


4/25
```

Previous Next

Figure 5.4: Browsing messages sent/received by node `c03-217a-51119`

5.2 User Stories

This section describes a few real life InfiSpector use cases.

5.2.1 Bad Coordinator Node

Every communication between servers has to have a node coordinator. User might have problems with communication and is not able to track coordinator. With InfiSpector he simply opens a view with BiPartite graph and spots which of the nodes is coordinator (sends majority of messages to all nodes) or maybe he can find out that there is none. With this information he is able to track error much faster.

5.2.2 Clogged communication

When communication is started, there is a huge communication traffic. After a while, communication should be stable. In other case, there is some problem. With our project, a user can easily look into time line diagram (Section 4.4) and find out when this occurs. If this information is not enough, he can also select this time period and look closely to communication with BiPartite chart. He can also browse each message flowing from one node to another and can spot the problem directly on UI without the need of opening textual Infinispan logs.

5.2.3 Adding New Node

Sometimes you need to add new node into already established cluster. With our tool, user is able to take a look on what exactly happens and with which nodes the new one starts to communicate and obtaining data from.

5.2.4 Performance

There were some performance issues with higher number of nodes. Chord diagram was too unclear and there were nothing to do to solve it. We are now looking for a replacement.

BiPartite diagram is not flawless too. With a high number of nodes the widening during node targeting is not smooth at all. This is why the possibility of groups was added. User can now select to join nodes to groups. Grouping takes some time, so the diagram is drawn a little later but there are no problems with lagging widening (if there is no more then 20 groups of course).

There is a delay between the click on a `Draw` button and graph displaying. There is no way to solve it because it takes some time to filter and return all these messages and create matrices for diagrams.

5.3 Discussing results with InfiSpector community

The results are presented to other members of the project team frequently. Also, the results are consulted with quality engineers from Infinispan team. Every time we got some constructive feedback or ideas we do our best to satisfy it.

For example recently we got feedback from one of the members of Infinispan team that filtering of messages would be nice. We have created possibility to create your own filter and display filtered message flow in new diagram.

Chapter 6

Conclusion

I was able to successfully design and implement core of the visual user-facing InfiSpector's interface with the help of D3js library and also connect our front-end to back-end. Diagrams are working perfectly and fit our needs for data visualization.

Firstly, I did a research of the rules of data visualization, useful libraries and some existing solution. I have found nice and useful diagrams which I have used in our project with a little adaptations so they fit our need more and connected them to our back-end. Also I had to design and create our very own bar chart which serves us as a time selector. To do this I have used a D3js library which is great for creation of dynamic elements. The last thing I've done is message browsing and grouping.

Hopefully, InfiSpector will be a useful tool for Infinispan community. We would like to improve InfiSpector so it fits more to Infinispan community. We would also like to modify it, so it could be used as a monitoring tool worldwide.

InfiSpector with newly added time line was already presented at DevConf. Also I have participated with this bachelor's thesis in Excel@FIT and 26.5.2017 it will be presented at Red Hat project day. Also, my code created during this thesis was already added into InfiSpector upstream repository.

Bibliography

[1] Aboukhadijeh, F.: *ZingChart*. [Online; visited 13.1.2017].
Retrieved from: https://www.componentsource.com/product/zingchart/about

[2] Anychart: *Line Chart*. [Online; visited 6.1.2017].
Retrieved from: http://6.anychart.com/products/anychart/docs/users-guide/Line-Spline-Step-Line-Chart.html

[3] Apache Software Foundation: *Apache Kafka*. [Online; visited 17.1.2017].
Retrieved from: https://kafka.apache.org/intro

[4] Bostock, M.: *D3js*. [Online; visited 12.1.2017].
Retrieved from: https://d3js.org/

[5] Chart.js: *Chart.js*. [Online; visited 13.1.2017].
Retrieved from: http://www.chartjs.org/

[6] DashingD3js.com: *D3.js Scales*. [Online; visited 29.4.2017].
Retrieved from: https://www.dashingd3js.com/d3js-scales

[7] DensityDesign Lab: *Raw*. [Online; visited 9.4.2017].
Retrieved from: https://github.com/densitydesign/raw/wiki

[8] Dutt, R.; Ödegaard, T.; Woods, A.: *Grafana*. [Online; visited 13.1.2017].
Retrieved from: http://grafana.org/

[9] Few, S.: *Tapping the Power of Visual Perception*. September 2004.
Retrieved from:
http://www.perceptualedge.com/articles/ie/visual_perception.pdf

[10] Google: *AngularJS*. [Online; visited 18.1.2017].
Retrieved from: https://docs.angularjs.org/guide/introduction

[11] Google groups: *Druid*. [Online; visited 18.1.2017].
Retrieved from: http://druid.io/docs/0.9.2/design/index.html

[12] OriginLab: *3D Bar Graph with Error Bar*. [Online; visited 6.1.2017].
Retrieved from:
http://originlab.com/doc/Origin-Help/3DBar-Graph-with-ErrBar

[13] Progress Software Corporation: *Pie Chart*. Online; visited 6.1.2017.
Retrieved from: http://docs.telerik.com/devtools/aspnet-ajax/controls/htmlchart/chart-types/pie-chart

[14] Red Hat: *Infinispan*. [Online; visited 14.1.2017].
Retrieved from: http://infinispan.org/about/

[15] Schlueter, I. Z.: *npm*. [Online; visited 18.1.2017].
Retrieved from: https://docs.npmjs.com/getting-started/what-is-npm

[16] The Chromium Authors: *Matplotlib Bar chart*. [Online; visited 6.1.2017].
Retrieved from: https://pythonspot.com/en/matplotlib-bar-chart/

[17] The Chromium Authors: *Promises*. [Online; visited 4.5.2017].
Retrieved from: https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Global_Objects/Promise

[18] tutorialspoint.com: *Node.JS*. [Online; visited 11.4.2017].
Retrieved from:
https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm

[19] Wikipedia: *Cartesian coordinate system*. [Online; visited 6.1.2017].
Retrieved from: https://en.wikipedia.org/wiki/Cartesian_coordinate_system

[20] Yang, F.: *Druid: A real-time analytical data store*. Online; visited 12.2.2017.
Retrieved from:
https://www.researchgate.net/figure/266656620_fig1_Figure-1-An-overview-of-a-Druid-cluster-and-the-flow-of-data-through-the-cluster

Appendices

Appendix A

CD content

- infispector/ – project InfiSpector
- infispector/infinispan_example_app – Infinispan example application that is ready to be run in multiple terminals, creating Infinispan cluster
- infispector/infispector_app – InfiSpector application for visualization data flows.
- infispector/kafka_druid_infrastructure – Helper files and configurations for setting up our Lambda Architecture: Zookeper, Apache Kafka and Druid.
- demo.mp4 – video demonstration over 4 nodes
- poster.pdf – presentation poster
- report.txt – report of my contribution
- README.txt – installation manual
- bp.zip – documentation in latex
- bp.pdf – thesis

Appendix B

Manual

This manual expects user to have *Druid version 0.8.3* and *Kafka* installed on his local machine.

1. Download code from Github[1].
2. Install project following installation manual
3. Start `skript.sh` and pass it path to *Kafka* and path to *Druid* as an argument.
4. Wait until script stops with `Firehose aquired!` (Figure B.1).
5. Change destination to `./infispector_app/`.
6. Start `grunt` by command `grunt`.
7. In your web browser go to `localhost:3000`.

Notice: Internet connection is necessary!

```
rConsumerConnector - [druid-example_c03-217a-1494250156763-2717205b], Committing
 all offsets after clearing the fetcher queues
2017-05-08T13:29:17,463 INFO [chief-InfiSpectorTopic[0]] kafka.consumer.Zookeepe
rConsumerConnector - [druid-example_c03-217a-1494250156763-2717205b], Releasing
partition ownership
2017-05-08T13:29:17,517 INFO [chief-InfiSpectorTopic[0]] kafka.consumer.RangeAss
ignor - Consumer druid-example_c03-217a-1494250156763-2717205b rebalancing the f
ollowing partitions: ArrayBuffer() for topic InfiSpectorTopic with consumers: Li
st(druid-example_c03-217a-1494250156763-2717205b-0)
2017-05-08T13:29:17,520 WARN [chief-InfiSpectorTopic[0]] kafka.consumer.RangeAss
ignor - No broker partitions consumed by consumer thread druid-example_c03-217a-
1494250156763-2717205b-0 for topic InfiSpectorTopic
2017-05-08T13:29:17,573 INFO [chief-InfiSpectorTopic[0]] kafka.consumer.Zookeepe
rConsumerConnector - [druid-example_c03-217a-1494250156763-2717205b], Consumer d
ruid-example_c03-217a-1494250156763-2717205b selected partitions :
2017-05-08T13:29:17,576 INFO [druid-example_c03-217a-1494250156763-2717205b-lead
er-finder-thread] kafka.consumer.ConsumerFetcherManager$LeaderFinderThread - [dr
uid-example_c03-217a-1494250156763-2717205b-leader-finder-thread], Starting
2017-05-08T13:29:17,580 INFO [chief-InfiSpectorTopic[0]] kafka.consumer.Zookeepe
rConsumerConnector - [druid-example_c03-217a-1494250156763-2717205b], end rebala
ncing consumer druid-example_c03-217a-1494250156763-2717205b try #0
2017-05-08T13:29:17,585 INFO [chief-InfiSpectorTopic[0]] io.druid.segment.realti
me.RealtimeManager - Firehose acquired!
```

Figure B.1

[1] https://github.com/infinispan/infispector

Appendix C

Poster

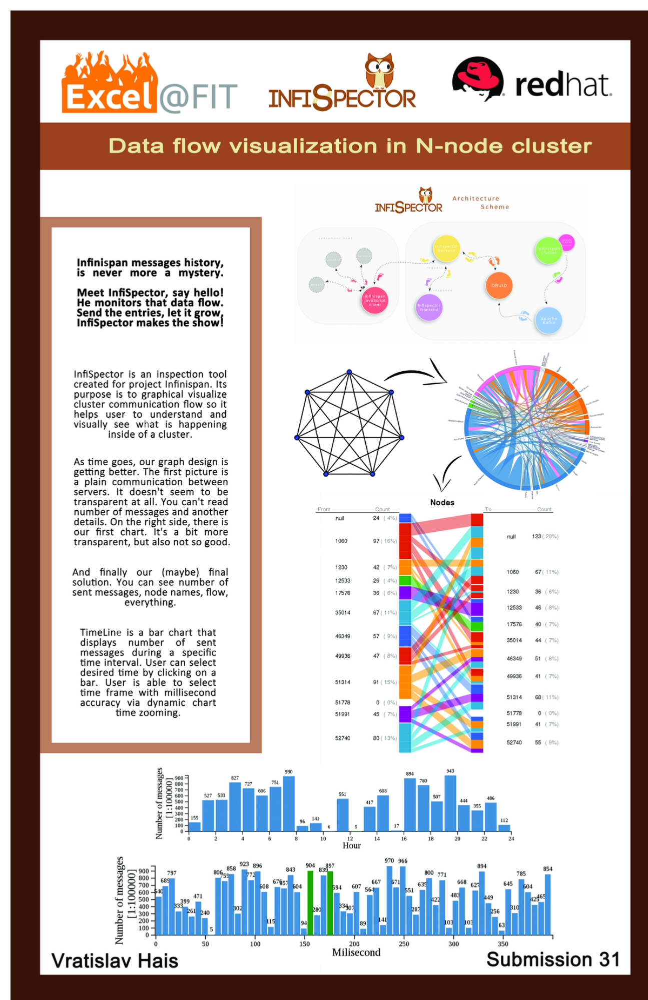


Figure C.1: Poster from Excel@FIT