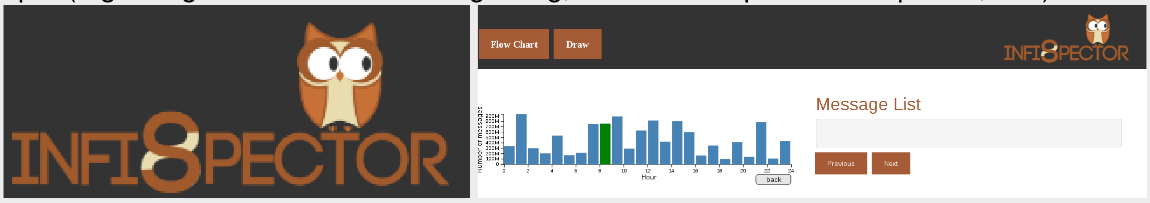
BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Data flow visualization in N-node cluster

Vratislav Hais*

Abstract
Project InfiSpector is developed with an intention to be a helpful inspection tool for Infinispan developers (NoSQL caching solution, distributed key-value store) during cluster debugging process. In a problematic situations, manual checking of huge textual logs is too demanding, time consuming and issues can be missed easily. InfiSpector displays cluster communication flows graphically so it helps users to understand and visually see what is happening inside of a cluster. Our primary goal is to help in situations where user could be able to see problematic cluster behaviour at the first spot (e.g. congested traffic at the beginning, no traffic in specific time period, etc.)



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

Supplementary Material: Demonstration Video — Downloadable Code

*xhaisv00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction and Motivation

For the further understanding of my work it is necessary to firstly say a few words about Infinispan. Infinispan is an OpenSource NoSQL, key-value, caching solution. The purpose of Infinispan is to expose a data structure that is distributed, highly concurrent and designed ground-up to make the most of modern multi-processor and multi-core architectures. It is often used as a distributed cache, but also as a NoSQL key/value store or object database [1].

Our idea is to create a tool that would greatly help Infinispan developers during their work while debugging communication message flows inside Infinispan cluster. There could be a wast number of messages sent in the Infinispan cluster daily, so it is a relevant and real-life problem to be able to efficiently detect a root cause for any kind of an error. Currently, there are no cluster monitoring tools tailored specifically for Infinispan needs and developers or testers are forced to check every message capture log manually. Logs can be huge (giga or terabytes of simple textual information) and that's definitely not a pleasant task to find an error there (better to say: a needle in a haystack). There is non-trivial time consumption for such debugging exercise to be completed and faults can be missed really easily.

This was a prime motivation for starting a new project under Infinispan ecosystem, called InfiSpector. Our main goal is to monitor every single message that is being sent in N-node Infinispan cluster, capture that message, store it for further analysis and display nice and helpful charting in our inspection tooling web user interface.

2. InfiSpector

InfiSpector is an open source project created by a team of students from FIT BUT which is led by Mgr. Tomáš Sýkora. Personally, I am responsible for a graph designing part. The goal of my bachelor thesis was to create a time line diagram, which will also serve as a time selector, and find a suitable solution for visualizing communication in N-node cluster.

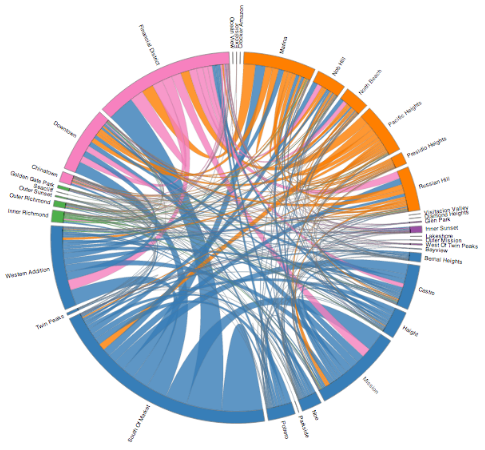


Figure 1. chord diagram

As there is no tool tailored directly for Infinispan specific messages it seems that InfiSpector is an ideal tool for such task. It's designed mainly for Infinispan, but can be used for any other project with similar problem. InfiSpector is able to transparently show communication between nodes. With newly added graph (Fig 2) you can even select a desired time window for detailed observation. The height of bars shows how many messages have been sent during particular time period. After the selection, 4 graphs with different filters are displayed to the user. It is possible to add various filters and even browse every single message sent by a specific node.

The principal advantage of InfiSpector is in its simplicity and possible general usage for similar problems – everywhere where you need to monitor internal cluster communication between N nodes ”talking” to each other.

3. InfiSpector's architecture

Core of InfiSpector's architecture[1] is built with the use of Apache Kafka and Druid.io. Apache Kafka is an open source project developed by the Apache Software Foundation written in Scala and Java. Kafka 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 [2].

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

Druid is an open source column store designed for online analytical processing queries on event 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 and is cost effective.
- Druid supports rolling updates so you are able to browse your data and use query even during software updates.
- Druid can handle trillions of events, thousands of queries and petabytes of data [3].

4. Visualization

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

The primary goal of charts in general is the simplicity. The more complex chart is, the greater is risk that it will be hard to understand it.

5. Existing solutions

Before we have started creating our own project, we have been looking for an existing solution. First we tried to find whole project which monitors data flow in cluster. We have found few projects, such as Ganglia[2]. This solution, unfortunately, does not work offline. Ganglia puts its client to cluster and provides monitoring using that client. Another disadvantage is that Ganglia is not able to monitor communication in N-node cluster. After a few days of searching we gave up and decided to create our very own project. After solving a few questions about technologies we have been discussing what framework to use to create our diagrams. There was a possibility to use Grafana[3] or Raw[4]. Grafana is great when you want to create dashboard or some basic chart, but it didn't fit exactly our needs. Raw was a little better. You can create diagram online and just copy created diagram (in SVG) to your web page. There are some preset diagrams from which you could choose and then design axises and other little thing. What it lacks is, that you can't design diagrams any further.

[1] Draft of InfiSpector architecture

[2] about Ganglia

[3] about Grafana

[4] about Raw

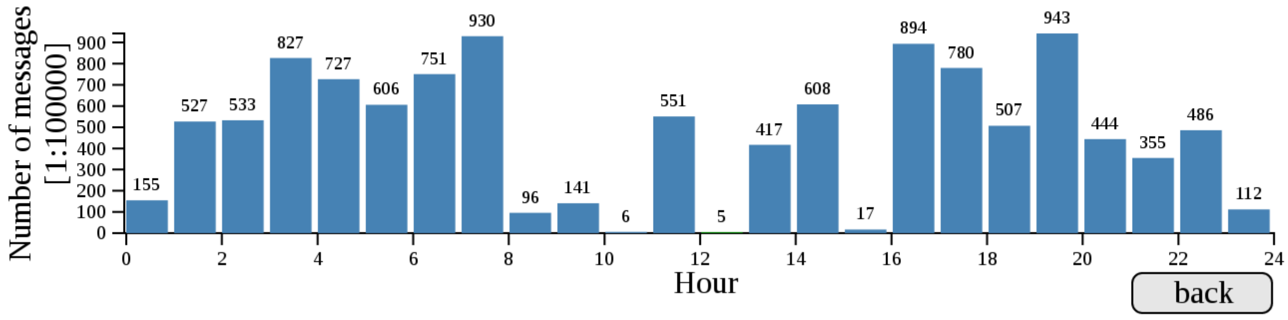


Figure 2. Newly added timeLine graph

6. Graph design

In InfiSpector we use several types of graphs. Every graph is programmed with the use of D3js framework.

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 [4].

In this section I will describe all graphs that are used in our project and that I have created or modified.

6.1 Time Line

This is the newest graph in InfiSpector. It is a bar chart that displays number of sent messages during a specific time interval. User can select desired time by clicking on a bar. Selected bar changes its color to red (Fig. 3). Interval selection (selecting second bar) leads to change of units with minimum and maximum value depending on selected interval. In the end, if needed, the user is able to select time frame with millisecond accuracy via dynamic chart time zooming.

6.2 BiPartite

BiPartite is a graph used to display actual communication between nodes. We have found this solution on D3js main page with examples and decided to use it. I have successfully edited code to use our data which required edition of existing functions and addition of on click functions. When user clicks on the node, all messages sent by this node are displayed in a box dedicated for this with possibility to browse messages one by one (Fig. 4). When mouse is over the node name label or lines displaying communication, section gets focus and is widened over the whole graph (Fig. 7).

Four such graphs are displayed at a time. Every graph have a specific filter (Fig. 6) that filters only message with a particular string in their internal content. User can also add his very own filters (Fig. 5) which will add new graphs on the page.

6.3 Chord diagram

Such as BiPartite (Sec. 6.2) Chord diagram is used to display communication between nodes. Chord diagram was our first working graph added to InfiSpector and was also found at D3js main page with examples. Only difference between BiPartite and Chord is their design (Fig. 1).

We have found that Chord diagram is unclear to users and it will be removed in the next project version. However, for demonstrative purposes works well.

7. Implementation

In this section I will describe implementation part of each diagram and connection between front-end and back-end.

Whole project is open-sourced with the use of github[5].

Diagrams are written in JavaScript with the help of D3js library.

7.1 Time Line

Time line diagram creates a huge part of my bachelor's thesis. It's my first true experience with creating a graph from scratch. As mentioned earlier, graph is created with use of D3js. D3js allows to use predefined functions which ease chart creation. Unfortunately, I could not use any predefined function because of the format of our data. This means I had to come up with conversion on spacing by trial-and-error method. Next thing to solve was to create an array containing values needed for spacing, width and height of bar and the number of messages. Number of messages in specific time are acquired in a for loop in which is used JavaScript method called promise[6]. Promises are used because we have to get values from Druid (2) and it takes some time so it is appropriate to work asynchronously.

Surprisingly the worst part for me were axes. It had taken a really long time before I discovered how to

[5]link to github can be found at the begging of this work
[6]About promises

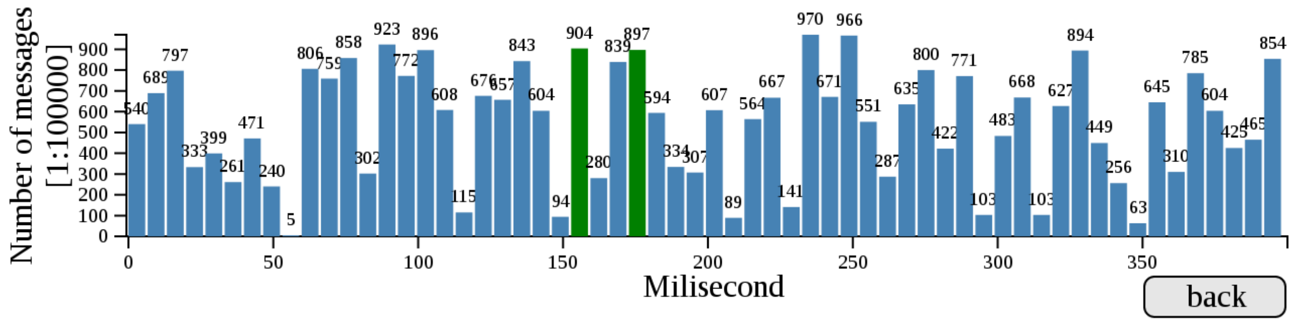
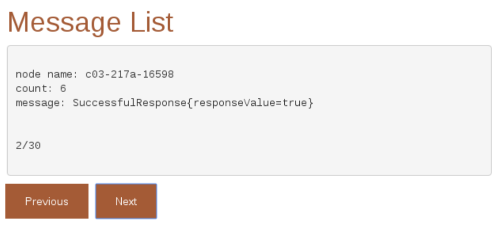


Figure 3. timeLine graph with selected bar



```
node name: c03-217a-16598
count: 6
message: SuccessfulResponse{responseValue=true}

2/30
```

Figure 4. box with sent messages

Figure 5. adding filter

narrow lines and turn text. Animations (color changing, text appearing) are pretty easy in D3js such as on click functions and mouse over functions. Lowering layers (in a meaning of units) was a little tricky. Here I had to use global variables for storing selected values, multiplier and past layers. I didn't came up with an idea how to change layers without removing whole diagram so during every layer change I have to remove diagram and recreate it.

The most tricky part was how to obtain selected time frame. The problem is, that user can select too wide interval, so it's pointless to change units. In that case, new multiplier (how many units represents one bar) is set and units stays the same. When getting selected time I had to take this into consideration. That is main reason why I use global variables. As written earlier I store multipliers, layers and values inside them and with this I am able to calculate and return selected time.

The final result can be seen at Fig 2.

7.2 BiPartite

BiPartite (Fig. 6) is an open source diagram which is available at D3js official page with examples[7]. Code and preview is here. This is a newer version than we are using.

This code had to be changed to fit our needs. Such as description, on click function, values and spacing between names of nodes and graph itself. We have to get node names and communication before drawing diagram. Getting node names is arranged by JavaScript promise which sends request to Druid. Druid returns an array of nodes. In the for loop we cycle through this array and create a JSON of these names. With this JSON and desired message filter we sends request to Druid to receive matrix with communication. This matrix is used as argument to draw BiPartite graph.

[7]D3js examples

7.3 Filters

Filters create a really big part of InfiSpector. With filters it is easier to monitor communication between nodes. Every session starts with 4 default filters. Users can easily add their own filters. All you have to do is to fill entry under graphs with one or more filters separated by comma and click a button. Another graph is added for every filter.

Clicking a button calls function that withdraw content of entry. Content is parsed with regular expression and stored in an array. We iterate through this array and calls function with current value as an argument. In this function we get communication matrix (as described in section 7.2) and add graph with that matrix.

8. User Stories

This section describes a few real life InfiSpector use cases.

8.1 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.

8.2 Clogged communication

When communication is started, there is a huge communication traffic. After a while, communication

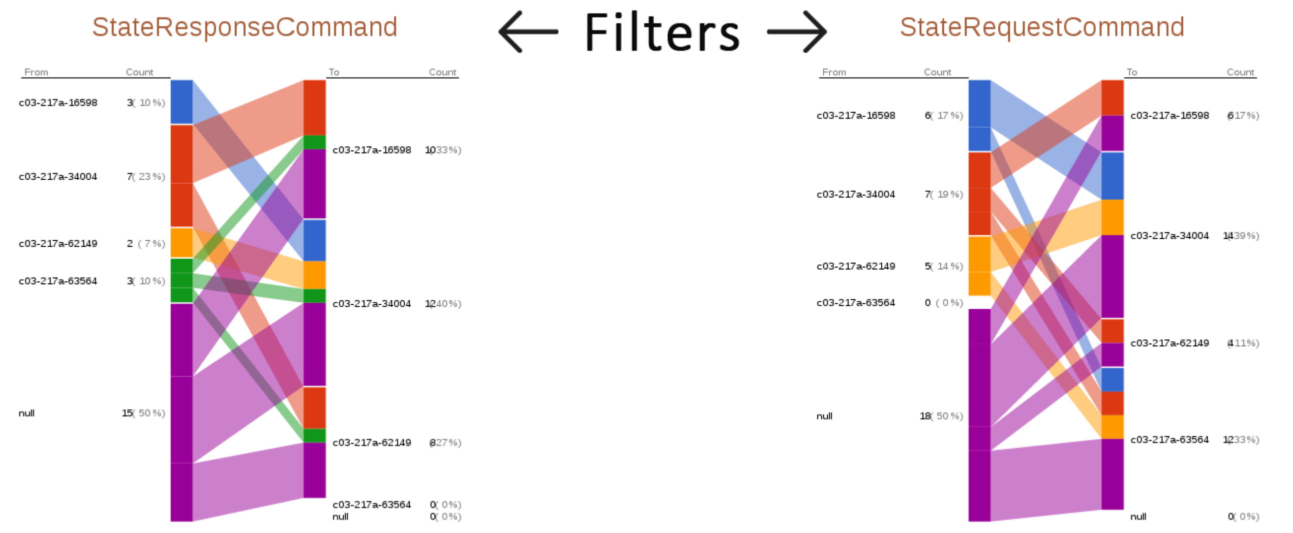


Figure 6. Two different filters. You can see that there is different communication flow

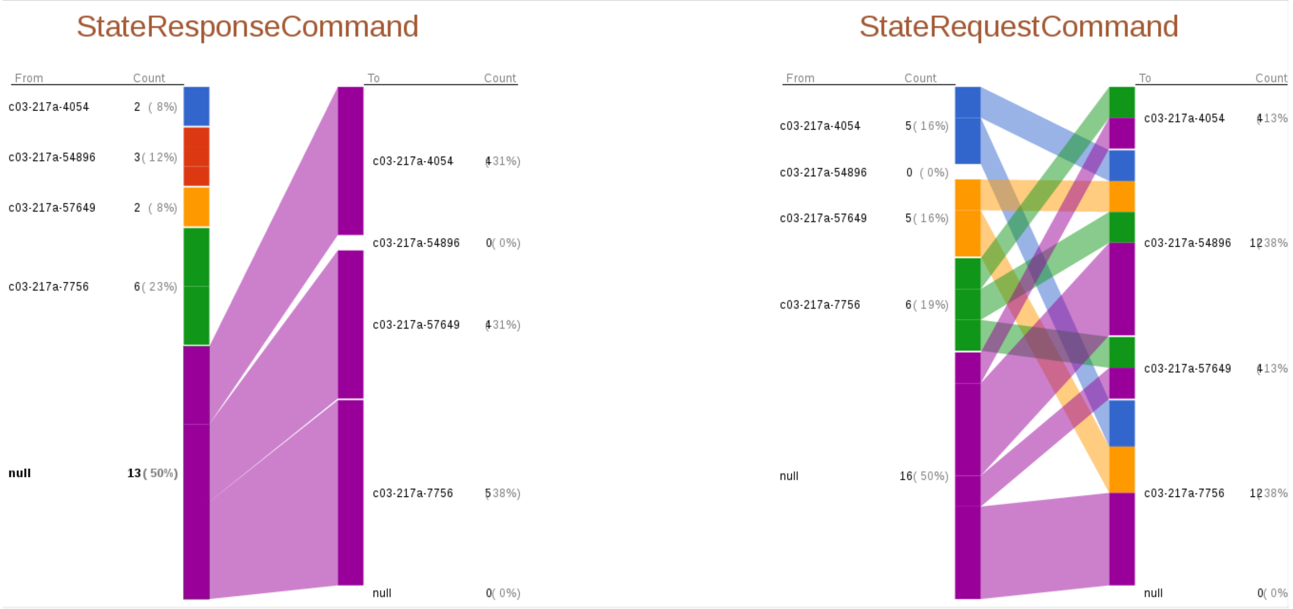


Figure 7. On the left side graph with focused node, on the right side graph with no focus

should be stable. In other case, there is some problem. With our project, a user can easily look into time line diagram (Sec. 7.1) 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.

8.3 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.

9. Future Plans

In the future I would like to modify the code of time line and add new higher layer - days. 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 (link at the main page).

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.

10. Conclusion

In this paper I shortly described Infinispan, our motivation to implement a helpful inspection tool, Infispector's architecture and visualization from general point of view. My own contribution is described mainly in sections 5 – Graph design, 6 – Implementation and 7 – User stories.

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

Acknowledgements

I would like to thank my supervisor Mgr. Tomáš Sýkora for his help on this work and for the opportunity to cooperate on InfiSpector development. It means a lot to me.

References

[1] Red Hat. *Infinispan*. [Online; visited 5.4.2017].

[2] Apache Software Foundation. *Apache Kafka*. [Online; visited 6.4.2017].

[3] Google groups. *Druid*. [Online; visited 6.4.2017].

[4] Mike Bostock. *D3js*. [Online; visited 5.4.2017].