Distributed D3: A web-based distributed data visualisation framework for Big Data

Xiaoping Fan

Department of Computing

Imperial College London

This dissertation is submitted for the degree of

Doctor of Philosophy

Data Science Institute 2019

Dedication

I would like to dedicate this thesis to my dearest parents.

Declarations

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This thesis is the result of my own work and all of the previous works mentioned in this thesis have been referenced and listed in the bibliography.

Acknowledgements

I would like to thank my supervisor Prof. Yike Guo, for his guidance, inspirations and advice on my personal development throughout my PhD study.

Secondly, I would like to thank to my second supervisor Dr. David Birch, for his patience, supports and suggestions in my research.

Meanwhile, I would like to thank my collaborators and colleagues at Data Science Institute, Mr. Evann Courdier, Mr. Guillaume Paillot, Dr. Jingwen Bai, Mr. Senaka Fernando, Dr. Miguel Molina Solana, for their helps and discussions during my PhD at Imperial College London.

I also would like to thank my department administrator Dr. Amani El-Kholy, for her understandings and supports during the hard times in my PhD study.

Finally, I would like to thank my parents, they have always encouraged me to overcome unexpected challenges and to develop myself to be a better person.

Abstract

The influx of Big Data has created an ever-growing need for analytic tools targeting towards the acquisition of insights and knowledge from large datasets. Visual perception as a fundamental tool used by humans to retrieve information from the outside world around us has its unique ability to distinguish patterns pre-attentively. Visual analytics via data visualisations is therefore a very powerful tool and has become ever more important in this era. Data-Driven Documents (D3.js) is a versatile and popular web-based data visualisation library that has tended to be the standard toolkit for visualising data in recent years. However, the library is technically inherent and limited in capability by the single thread model of a single browser window in a single machine, and therefore not able to deal with large datasets.

In this thesis, the main objective is to overcome this limitation and address possible challenges by developing the Distributed D3 framework that employs distributed mechanism to enable the possibility of delivering web-based visualisations for large-scale data, which also allows to effectively utilise the graphical computational resources of the modern visualisation environments. As a result, the first contribution is that the integrated version of Distributed D3 framework has been developed for the Data Observatory. The work proves the concept of Distributed D3 is feasible in reality and also enables developers to collaborate on large-scale data visualisations by using it on the Data Observatory. The second contribution is that the Distributed D3 has been optimised by investigating the potential bottlenecks for large-scale data visualisation applications. The work finds the key performance bottlenecks of the framework and shows an improvement of the overall performance by 35.7% after optimisations, which improves the scalability and usability of Distributed D3 for large-scale data visualisation applications. The third contribution is that the generic version of Distributed D3 framework has been developed for the customised environments. The work improves the usability and flexibility of the framework and makes it ready to be published in the open-source community for further improvements and usages.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivations

The Data Science Community has been growing significantly in order to address the influx of Big Data. These data sources range from Internet of Things sensor networks to the growing Open Data movement [1] as well as the Data Stores holding them [2]. Such new data streams are leading to the global economy's economic benefit using a new data economy and start-ups that result in new insight from data. A data-driven economy succeeds because of the insight generated from data being shared. At the heart of this is data visualisation. Indeed, visualisation and visual analytics continue to be major tools for generating insight from data. Recently, high impact infographics and data-driven story telling powered by new streams of Big Data and Open Data have been observed to be growing.

On the other hand, the existing visualisation techniques remain less affected by the rise of Big Data, with the majority of visualisation tools and environments having the ability to show only small datasets and very few data points. For this reason, it is important to implement coarsening and aggregation techniques for presenting the data as it prevents practitioners from gaining a complete view of the data and reduces the ability to permeate into the data for examining the data trends in detail. A key role for universities is leading the development of tools for better Data Science. Further, within the Data Science Community, the Data-Driven-Documents (also known as D3.js) [3] methodology and library has become the standard tool for visualising data. In fact, the open-source library has more than 15,000 collaborators [4] and is considered the standard tool for several companies including New York Times [5] and the BBC.

However, D3.js library is technically inherent as it has limited capability and is unable to address large datasets, a few thousand data points being the maximum it can process. Essentially, the approach is limited by an out-dated single machine and single-threaded model in JavaScript [6]. At the Data Science Institute, a distributed model is being developed for data visualisation based on the D3.js approach. This will include the same advantages of distributed computation techniques that underpin the Big Data processing community, including Hadoop [7] and Spark [8], while implementing them to the visualisation community. Therefore, this can ensure easy scalability of visualisations in terms of multiple computers, high-resolution screens, and video wall environments that are becoming increasingly common, and it will also enable orders of magnitude more data points to be visualised for greater insight.

Furthermore, the potential impact of the realisation of Distributed D3 on a modern visualisation facility can be significant. The empowered large visual space not only provides a large high-resolution visualisation at a time, but also enables multiple times of raw data points to be plotted and visualised on the tiled screens all at once. This allows finding subtle patterns in the large datasets, and therefore helps researchers to identify trends and make predictions based on real insights rather than assumptions. In addition, the interactive features of Distributed D3 can further improve the transparency and visibility of the large and growing amount of data, which further allows the high-volume data flow to be monitored in a real-time manner, and hence providing opportunities to control and respond immediately on certain events, such as errors or malfunctions in a system, as well as the potential risks.

1.2 Objectives

The main objective in this thesis is to deliver a robust Distributed D3 framework that is able to preserve the API compatibility of D3.js library for the simplicity of use. As a result, the framework is expected to resolve the performance bottleneck of D3.js which prevents the visualisation for large-scale data, and therefore improves the scalability and usability of D3.js in general. The framework should further enable a variety of visualisation environments to be configured and programmed by a wide community of developers for the collaboration and research purposes.

The work can be divided into the following stages based on the priority of its development,

- Integrated version of Distributed D3 for Data Observatory, which should reveal the possibility of utilising distributed approaches to overcome the performance bottleneck of D3.js, and therefore allow developers to evaluate and utilise the framework on the Data Observatory.

- Evaluation and optimisations of Distributed D3 with an emphasis on application, which should further improve the framework by investigating and addressing the potential performance issues, and thus optimise the framework for visualisation application with large-scale data in terms of scalability and usability.

- Generic version of Distributed D3 for a wide range of communities, which should allow the framework to be configured and programmed on the customised visualisation environments, and it should also provide the flexibility on the configurations of its components to enable the framework to be widely used and adapted for a variety of purposes.

Key Challenges

The main challenges in the development of Distributed D3 are to build a distributed system that allows distributed rendering and distributed data to be realised for both static and dynamic visualisations, as well as to build a robust yet flexible underlying distributed communication network.

- Distributed Rendering, apart from the static distributed rendering that can be realised by knowing the positions of a segmented visualisation, the dynamic distributed rendering for a smooth and synchronised animation across displays can be challenging.
- Distributed Data, the main challenges can be expected to deal with the data segmentation for various types of chart, and also to allow transferring the data segments to the targeted displays on demand, such as in an animation.
- Distributed Network, the classic client-server model is expected to resolve the most of communication issues in framework initialisation and animation synchronisation. Whereas it becomes more challenging if we would like to build a pure peer-to-peer network while addressing these issues.

1.3 Related Works

The majority of the existing research focus on distributed displaying and parallel rendering [9-25]. As far as this research work is concerned, no other equivalent frameworks have been developed thus far. This section will review the developments of the relevant existing frameworks and their main features, and then summarise these visualisation frameworks as a designing reference for the present study.

For several years, the computer graphics community has been interested in using high resolution images and high-performance rendering pipelines for examining large datasets and for preparing these images, respectively. Whitman [9] evaluated how, from the late 1970s, numerous attempts have been made to parallel rendering. The classification of parallel rendering by Molnar [10] in the mid-1990s was instrumental in setting the precedence for numerous innovative approaches concerning dynamic rendering of high-resolution graphics. On the other hand, it was only recently that the visualisation of these high-resolution graphics in tiled display walls became popular. It was 15 years ago that examples including SAGE [11], Chromium [12] and Parallel-SG [13] were put forth. Such early examples led to the development of modern Scalable Resolution Display Environments (SRDEs) along with different infrastructures introduced during the previous 5 to 10 years. Although such systems are varied, they use different strategies for resolving similar problems, and they may not always be appropriate for all problems that need an SRDE.

Despite the fact that the SRDEs' middleware standardisation is yet to be accomplished, multiple authors have applied various ways trying to categorise them. For example, Chen et al. [14] considered their execution model to classify them into two groups, client–server and master–slave. Ni et al. [15] considered their data distribution architectures in terms of the distributed rendering software and display data streaming software to classify them. Chung et al. [16] took into consideration which applications they focus on and classified them into four groups of transparent frameworks

concerning legacy applications, interactive application frameworks, distributed scene graph (DSG) frameworks concerning 3D graphics applications, and scalable rendering frameworks. Further, Renambot et al. [17] considered their deployment models to classify them into browser-based and desktop application. Considering such classifications of SRDEs' middleware, the present study began by examining window management systems. Such systems offer an integrated workspace to visualise distributed data so that various applications can be executed at the same time throughout numerous tiled displays. Moreover, Distributed Multi-head X (DMX) [18] was developed for offering multi-head support to the system's X-Windows desktop that includes numerous displays. A client–server model is implemented in DMX such that the server node dispenses the visual elements that are then rendered in client nodes.

Scalable Adaptive Graphics Environment (SAGE) [11] is also a well-known window management architecture, with SAGE2 [17][19], its second version, being available and called Scalable Amplified Group Environment. This second version, SAGE2, is a browser-based client–server cross-platform middleware that is distributed using various useful applications that aim to effectively resolve remote collaborations in data-intensive environments. Both these systems have become substantially popular and are used extensively on a global scale. DisplayCluster [20], although not as favoured as SAGE2, is also a dynamic windowing environment that includes built-in capabilities that can help view media and aids in streaming and displaying ultra-high-resolution images as well as video content. Although these are transparent frameworks that have a non-invasive programming model, they are extensively dependent on pixel streaming for the distribution of the majority of server-to-client visual information. Because of this, these systems' scalability is significantly decreased concerning supported resolution and their applicability to environments with high-performance networks is also limited.

Parallel graphics rendering middleware including the Image Composition Engine for Tiles (IceT) [21], compared with window management approaches, distribute the rendering workload throughout numerous distributed system's nodes. Because of this, applications can classify the display area into tiles and allocate them to one or more processors for executing sort-last parallel rendering. Further, every processor can simultaneously render content for multiple tiles. IceT allocates the rendering of M tiles' graphical content throughout N processors and implements numerous strategies including map-reduce and binary trees for devising the resulting tiles as well as producing a distinctive image. Despite the fact that IceT fulfilled the anticipated performance as well as scalability requirements, it needs visualisations for it to be purpose-built according to its rendering pipelines, thereby restricting its scope.

Chromium [12] is a framework developed according to the older WireGL [22] system from Stanford, which offers a distinct approach. Chromium manipulates and distributes streams of graphic API commands on different computers' clusters. It can execute OpenGL-based applications as it intercepts OpenGL commands that are disseminated to client nodes. The commands that such clients receive are executed for rendering their corresponding part of a larger picture. It should be noted that Chromium has a major drawback of suffering from high network usage, even in cases where there is no change in the scene, caused by its low-level focus that prioritises precision rather than performance. To address such drawbacks of Chromium, Garuda [23] was introduced which decreases the consumption of network bandwidth as it gathers and manages the transmitted geometry at rendering nodes. To ensure that only the necessary aspects of the scene are transferred to every client, it implements an adaptive algorithm through which the scene graph is culled to frustums hierarchy for assessing the objects that can be seen in every tile of the wall. Although Garuda fulfilled several needs of the present study, there are certain performance implications including the requirement of a high-end server machine as well as a gigabit ethernet for ensuring a stable frame rate for animations. A significant limitation of Garuda, however, is that it is capable of

supporting applications of only a scene graph type, thus reducing its applicability in collaborative visual analytics.

Equalizer [24] is a toolkit aimed at scalable parallel rendering that follows OpenGL. Equalizer offers an application programming interface (API) for devising scalable graphics applications which can function in various configurations such as tiled displays. Moreover, the Cross-Platform Cluster Graphic Library (CGLX) [25] offers a different API which enables same copies of an OpenGL-based application to be executed on all clients as well as visualisation data concerning all the clients to be replicated. Though the development of CGLX aimed to improve Chromium, it offers a significantly broader scope than Garuda and is not limited to scene graph type applications, such as Equalizer. Compared to Equalizer, however, CGLX can be easily maintained and offers more transparency. On the other hand, scalability is a major limitation of CGLX. It was observed that CGLX's performance (assessed in FPS as is common for animated content) decreased with an increase in the number of display tiles because of the synchronisation overhead resulting from the head node. Hence, compared to displays with several screens, animations would run considerably faster on those displays that had fewer screens.

To summarise the main literature findings above, in the field of distributed displaying and rendering, DMX was the first distributed display environment that enabled a fully-functional windowing environment across a cluster. The SAGE was developed with an architecture that had all pixels streamed to the multi-node display from sources over the network, which is more focused on scalable rather than environment in comparison with DMX. As opposed to SAGE's streaming architecture, CGLX provides a semi-transparent OpenGL-based graphics framework for distributed visualisation systems. For such rendering environments, the major problems often involved the limitations of scalability, which have been significantly enhanced in the later works of SAGE2 and DisplayCluster.

Moreover, the key features of the main distributed visualisation systems and frameworks can be summarised as follows: Windowing environment for multiple applications, which is a centralised system that places windows with content (applications, multimedia) from various sources on display in the visualisation system. Client-server architecture, in which several clients pull requests from a server while also receiving pushed data from it. Extensible 2D and 3D applications, which provides a possibility for developing and adjusting the existing applications for system integration. Off-the-shelf application support, which is a method of sharing the application screen and user interface concerning applications that cannot be changed. Multi-user interaction and remote collaboration, which helps multiple users use the system interactively and remotely. Content with unlimited resolution, wherein content need not be limited to any fixed maximum resolution. Cloud-based infrastructure, which is the ability to host a server in the cloud so that clients can access this system.

Finally, by reviewing these existing visualisation systems and frameworks, it was noted that frameworks which support dynamic windowing environments rely on pixel streaming offered inadequate performance. Meanwhile, frameworks focused on DSG were limited and had issues with transparency, and other frameworks were limited in their scalability and thus may not be used in SRDEs having varied dimensions. Therefore, Distributed D3 is designed with a different approach that aimed to resolve such limitations, with its design being influenced by and borrowing from several primary concepts of the existing frameworks.

1.4 Thesis Structures

The structure of this thesis is arranged as follows,

- In Chapter 1, we specify the motivations and objectives of the work, and we discuss the related works that are relevant to this research. An overview of the thesis structure is then provided with a summary of contributions.

- In Chapter 2, we discuss the definition and meaning of data visualisation. We provide the relevant background of data visualisation, web technologies and data-driven documents, and also review the possibilities of building a distributed system. In the last section, we also include an overview of the data observatory with case studies.

- In Chapter 3, we propose the integrated version of Distributed D3 framework for Data Observatory. We first compare the possible approaches in design, and then detail the design and implementations of the framework. The implementation results with examples are demonstrated, and the benchmarking results are discussed before the conclusion of this chapter.

- In Chapter 4, we propose the optimised and upgraded version of Distributed D3 framework for applications with large-scale data. We first investigate the bottleneck of the existing Distributed D3, and then optimise the framework by addressing the underlying issues and also upgrading the support for newer D3. A demonstrating application is illustrated, and the benchmarking results are discussed before the chapter conclusion.

- In Chapter 5, we propose the generic version of Distributed D3 for the customised environments. We first illustrate the detachment and serverless design of the framework. We then discuss the implementations that include a new network interface and two detailed distributed approaches. A customised

visualisation environment is also set up for the testing purpose. The benchmarking results of the generic version are discussed and compared with the previous version.

- In Chapter 6, we conclude the current development of the work and identify the remaining challenges and issues, and we also discuss the potential future works in the last section.

1.5 Contributions

In this thesis, we develop the Distributed D3 framework that addresses and resolves the performance bottleneck of the D3.js library by implementing the distributed mechanisms. The realisation of this work enables the possibility of delivering web-based visualisations for large-scale data, by effectively utilising the graphical computational resources of the state-of-art visualisation environments, such as Data Observatory. The work further enables such possibility to be realised on a variety of customised visualisation environments that can be configured and programmed by a wide community of developers.

Specifically,

- We present an integrated version of Distributed D3 for Data Observatory. The result shows to overcome the performance limits of the D3.js that proves the concept of Distributed D3. The improvement of the overall performance and scalability further enables a wide community of developers to work on large-scale data visualisation in the Data Observatory.

- We present an optimised and upgraded Distributed D3 for the large-scale data visualisation. The test result demonstrates to increase the overall performance by 35.7% compared with the previous version, and the support of newer D3.js further extends the functionality of the framework. The version thus improves the scalability and usability of the Distributed D3 for visualisation applications with large-scale data.

- We present a generic version of Distributed D3 for customised environments on demand. The test result shows the version is light-weighted and faster than its ancestors, and it is also featured by its flexibility of switching networks between classic client-server and pure peer-to-peer implementations when needed. The

version is ready to be published for the open-source community for further impact and improvement.

Chapter 2 Background

"Visualisation gives you answers to questions you didn't know you had."

– Ben Schneiderman

2.1 Web Data Visualisation

2.1.1 Introduction

The rise of Big Data has led to an increasing need of analytic tools to obtain insights from the increasingly large datasets. Visual perception is a primary tool of humans for retrieving information from the outside world and thus has the distinguished ability to rapidly differentiate patterns in a pre-attentive manner [26]. Hence, visual analytics are crucial for data analysis.

Visualisations has to be seen to be considered truly visual [27]. Hence, it is necessary to ensure that a piece of work can be seen by others, for which internet publication is the fastest method for distributing information globally. Collaborative work with web-standard technologies helps the work to be made visual and to be seen by anyone using a web browser, because of that the operating system as well as device type (i.e. laptop, desktop and smartphone on Windows, Mac and Linux) can be flexible.

Data-Driven Documents (also known as D3.js) is a popular web-based data visualisation library that helps in generating dynamic and interactive data visualisation

in diverse graphical forms. The standardised representation of D3.js improves the expressiveness as well as accessibility, while providing significant performance improvements and enabling transitions to be animated. Performance benchmarks can further help D3.js to be demonstrated as at least two times faster compared to its ancestor [3].

2.1.2 Data Visualisation

2.1.2.1 What is Data Visualisation?

Data visualisation is the process in which graphical representations are used for communicating information. Before the written language was formalised, pictures were used for communication as images perform in parallel with the human perceptual system. Text analysis, however, is a sequential process that is restricted by the speed of reading [26].

Data visualisation can be perceived as both an art and a science [28]. While several consider it a branch of descriptive statistics, others regard it more as a ground theory development tool. Internet activity and increased number of sensors have led to the development of Big Data. The key challenges for data visualisation involve how this data can be processed, analysed, and communicated [29].

Data visualisation is the representation and presentation of data that exploits a viewer's visual abilities for amplifying cognition [30].

- The **representation** of data refers to how one decides to present data in a physical form, which may be a bar, a line, or a circle. This results in using the data as a raw material and generating a representation that presents its key features in the best possible way.
- The **presentation** of data extends beyond data representation and focuses on how data representation is implemented into the overall communication system, such as the colours, annotations, and interactive features.
- The exploitation of **visual perception abilities** concerns the scientific understanding of how the human brain and eyes perceive information in the best possible way. It focuses on the way in which individuals' abilities can be

harnessed using spatial reasoning, pattern identification, and big-picture thinking.

- **Amplifying cognition** concerns the optimising of a persons' ability to efficiently and effectively process the information and convert it into thoughts, insights, and knowledge. Data visualisation primarily aims to provide the readers with the feeling that they are more knowledgeable about a particular topic.

2.1.2.2 Why Data Visualisation?

There are various reasons for which data visualisation is important. The most apparent one is that the sense of sight is fundamental to obtain and understand information as humans are visual beings [26]. Using an effective visualisation can help a viewer to better analyse and reason data and evidence. Data visualisation simplifies data that is complicated and renders it more understandable and usable.

Furthermore, considering the nature of the human brain, charts and graphs are the best way of visualising large numbers of complex data in a simple manner. This is easier than reading spreadsheets or reports. Data visualisation is quick and easy and helps present concepts in a universal manner, enabling minor adjustments to experiment with different situations.

Visualising information helps in telling a story. It is a primitive communication technique, dating as far back as 30,000 BC in the form of cave drawings. Even before the development of written language in 3,000 BC, vision was a crucial means of communicating. With time, new ways of visualising information were developed. Today, although people have become familiar with basic charts such as line chart, bar chart, and pie chart, they rarely stop to think about the existing issues and how they can be improved.

It is important to note that data visualisation is an especially important discipline in the modern age. There is a digital consequence to almost everything that people do, with everyone's lives being consistently recorded and quantified. Although this may appear scary, the sharing of large quantities of information can generate exciting new opportunities for those who possess analytical curiosity and intend to explore the world. Data is unarguably an invaluable asset and has become so powerful that it can change the world for the better. If data is the oil, then data visualisation is the engine that facilitates its true value, which is why it is such a relevant subject to explore nowadays.

2.1.2.3 Categories of Data Visualisation

Data visualisation can be divided into two main categories: exploration and explanation [31]. These two categories have different purposes, and thus use unique tools and approaches that may not necessarily be suitable for the other. Hence, it is crucial to understand their differences to ensure that the correct approach is used for a specific visualisation task.

Exploration

Exploratory data visualisation is suitable in case of a large dataset, when one is unsure of its contents, and when needing insight into the data set. Its visual presentation can help in quickly identifying its features, such as important curves, lines, trends, or anomalous outliers.

Usually, the exploration is best conducted with high level of granularity. The dataset may contain lots of noise, but oversimplifying or removing too much information may cause something crucial being missed. Such visualisation is a fundamental aspect of data analysis and can be used to reveal the dataset's story.

Explanation

Explanatory data visualisation is best used when one already knows the data contents and attempts to share them with other parties, such the group head, a grant committee, or the public. Regardless of the audience, the individual thus knows the story from the beginning and can make designs that are exclusive to the story being presented. Hence, the individual must make editorial decisions regarding which information to keep and which to remove for being too distracting or irrelevant. For this, the focused data that can support the story being told should be selected.

If the exploratory data visualisation is part of data analysis phase, then explanatory data visualisation will be part of the presentation phase. This type of visualisation can occur alone or as part of a larger presentation such as a speech, a newspaper article, or a report. For explaining things in more depth in such situations, a supporting narrative is typically provided.

The Hybrid

A hybrid category can also exist involving a curated dataset, for which the data presented can help the reader conduct exploration to an extent. Such visualisations are often interactive, such as involving a type of graphical interface that enables the reader to select specific parameters. They can, therefore, determine insights in the data for themselves. Further, these insights may also not have been identified previously by the visualisation's creator.

2.1.2.4 Methodology of Data Visualisation

The common definition for taxonomy originates from biological sciences and typically separates organisations into groups of members that share similar characteristics. In this case, various charts function as the members while the shared characteristics are the functions of the primary data.

The appropriate visualisation methods are determined by the definition used when developing the methodology. This is crucial when clarifying the intention of your visualisation communication. The key communication purposes of all the classification methods are summarised below [30]:

- **Comparing categories** helps make comparisons between the relative and absolute sizes of categorical values. For example, a bar chart.
- **Assessing hierarchies and part-to-whole relationships** help in breaking down categorical values as per their relationship with a group of values or in presenting them as constituent elements of hierarchical structures. For example, a pie chart.
- **Presenting changes over time** helps exploit temporal data so that changes in patterns can be depicted throughout a particular timeframe. For example, a line chart.
- **Plotting connections and relationships** help in assessing relationships, trends, and distributions in multivariate datasets. These tend to involve intricate visual solutions that focus on allowing exploratory analysis. For example, a scatter plot.
- **Mapping geo-spatial data** aids in presenting datasets that possess geo-spatial properties through various mapping frameworks. For example, a choropleth map.

2.1.2.5 Data Visualisation on the Web

Traditional static visualisations can only present precomposed data views, but multiple static views are often necessary for presenting various perspectives concerning the same information [27]. Moreover, there is a limited amount of data dimensions if all visual elements are shown on the same surface at the same time. Depicting multidimensional datasets fairly using static images is very difficult. Hence, using a fixed image is the best method if differing views are not necessary to create a static medium, such as a print.

People can self-explore data using dynamic, web-based interactive visualisations. This is applicable to numerous interactive visualisation tools and has remained relatively unchanged since the initial introduction of 'Visual Information Seeking Mantra' by Ben Shneiderman of the University of Maryland in 1996 [32]. For the majority of contemporary interactive visualisations, an overview is first conducted, followed by zoom and filter and then details-on-demand. Using multiple functions is effective for ensuring that the data can be accessed by various audiences, from mere browsers to those seeking answers to specific questions.

The interactive visualisation can present an overview of data using tools best suited for 'drilling down' the details, and can help simultaneously address numerous tasks. This helps in addressing the problems of audiences, from those who have little current understanding of a new subject matter to those with an excellent grasp of the information. Interactivity also promotes engagement with data in ways that static images cannot. Moreover, animated transitions and well-designed interfaces can make data explorations feel like playing a game. Hence, interactive visualisation is an effective method of engaging people who are otherwise uninterested in the topic or data.

2.1.3 Web Technologies

2.1.3.1 HTML and CSS

Hypertext Markup Language (HTML) was first introduced in 1990 by Tim Berners-Lee during his time at the high-energy physics research centre CERN, which is the European Laboratory for Particle Physics. It was originally designed for scientific and engineering purposes. For several years after its invention, even with many amendments, it was still possible to detail the aspects of the language in a concise document (W3C-HTML-HIST). By November 1992, in addition to the previously mentioned title and paragraph components, HTML incorporated only aspects to enable the creation of hyperlinks, headings, basic lists, glossaries, examples or text that utilises monospace fonts and preserves any white space, and address blocks, which are usually in italics and comprised of the authors' information. Beyond this, there was nothing else at this stage.

Extensible Markup Language (XML) was then introduced by the W3C in February 1998. It is a limited form of Standard Generalised Markup Language (SGML). SGML has a greater degree of generality than XML, but the latter nonetheless is capable of defining syntaxes for languages, including HTML. A number of HTML versions have been delineated using XML instead of SGML. These are referred to as XHTML languages, the first of which was XHTML 1.0. Semantically, there are no differences between XHTML 1.0 and HTML 4.01; while syntactically, they are identical aside from a few minor limitations of HTML's generality in XHTML.

Document Type Declarations

Every specification of HTML includes a declaration that can be employed at the start of documents that are going to follow the specification. The HTML 4.01 and XHTML 1.0 specifications have three *flavours*. Each of these has a unique document type declaration, characteristics, and components, and they are as follows,

1. The *Strict* version which evolved in late 1997, when W3C was addressing more reliable parsing of HTML documents.
2. The *Transitional* flavour which is a superset of *Strict* HTML and contains deprecated elements and attributes. In other words, while the *Strict* version is considered best practice, the *Transitional* version possesses elements and attributes which should be avoided whenever possible since they can potentially be removed from HTML recommendations in the future.
3. The *Frameset* version constitutes a superset of a *Transitional* flavour. It incorporates a feature which permits multiple sub-windows or frames to be displayed within the client area of a web browser.

The following are the advised document type declarations for XHTML 1.0 Strict, XHTML 1.0 Frameset, and HTML 4.01 Transitional:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<!DOCTYPE HTML
PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

Unrecognised Elements and Attributes

At times, new web developers can become puzzled by another aspect of HTML, which is that browsers do not highlight unrecognisable element or attribute names contained in a document. This differs from the norm of programming, as if a keyword is mistyped in a Java program for instance, the error will be flagged, causing the program to stop running. However, the browser will try to display an entire web page, even if, for example, an element name ‘p’ is mistyped. In the case of attribute names that are not recognised, the browser simply ignores the attribute, whilst in the case of element names

that are unfamiliar, the browser will show the elements' contents as if the mark-up was absent.

The Cascading Style Sheets

It is possible that both a documents' semantics and presentation can be represented by using the HTML markup. However, it is recommended that this should mainly be the case for the documents' semantics, whereas the presentation of information in a document is best determined by using a different mechanism. One such mechanism is Cascading Style Sheets (CSS), which is a style sheet technology that can be utilised with both HTML and XML documents.

CSS Features

The primary purpose of style sheet technology is to distinguish the presentation of information from the main content of the information and semantic tagging. There are many benefits to doing so; one of which is that it enables the unamended presentation of the documents' information in several ways. This is in evidence in user selectable alternative style sheets. However, CSS has additional functions. For instance, the link element defines a media attribute that can be employed to identify the forms of media for a style sheets' design, in which amongst other aspects, could refer to printer output or the content to be displayed on a specific monitor or screen.

Style Inheritance

Cascading style sheets are structure-based, however in contrast, inheritance is determined by the documents' tree structure. Essentially, this means that a component

inherits a value by examining whether a property value already exists in the parent element in the document. If this is the case, the parents' value is inherited. Similarly, the parents' property value can be inherited from its' parent and so forth. For instance, taking id value theValue, an element will search its ancestral tree from its parent upwards until it reaches an element with a property value or the root HTML element. Figure 2.1.1 presents a model of this cascading mechanism [33].

When the search finds an element with a property value, that value will be taken by theValue as its property value. If the search of the ancestral tree returns no result, then ultimately the property will be given a value as per the CSS specification, and this is referred to as the property's *initial value* [34]. This is reasonable if you consider that an initial value is assigned to each element property upon the first reading of the document, which is subsequently amended if a value is later found to be supplied either by the cascade or the inheritance mechanism.

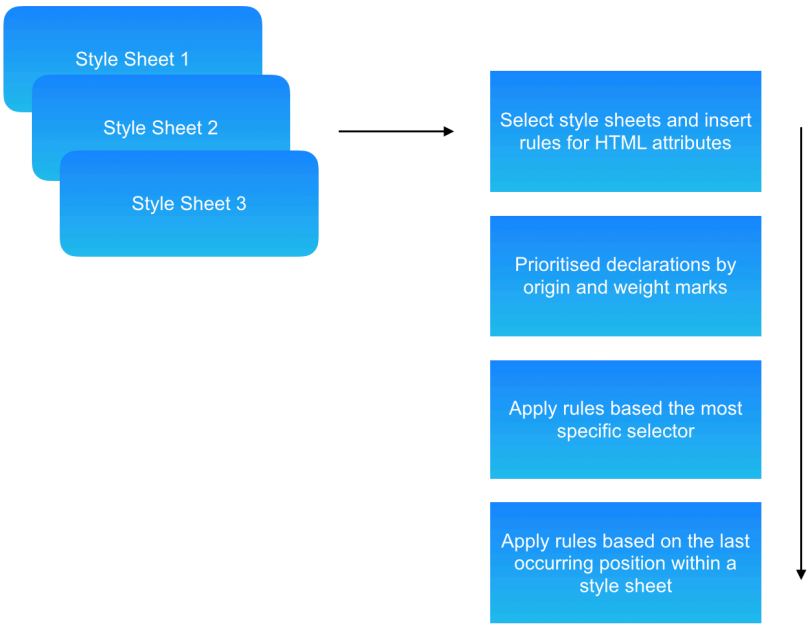


Figure 2.1.1 The stages of CSS cascading mechanism

Another significant aspect of inheritance is that although many CSS properties are inheritable, some are not. Usually, whether a property is inherited is apparent to the user based on their intrinsic knowledge. For instance, it is reasonable that an elements' height property is not inherited from its parent, as frequently, the parent has numerous children on numerous lines, and hence, children would normally have lower height.

2.1.3.2 JavaScript and Objects

JavaScript, as a component of the Netscape 2.0 release, was first developed by Brendan Eich [35]. During its preliminary phase, the language was referred to as LiveScript, however, prior to the final release of Netscape 2.0, there was a public announcement on December 4th, 1995 revealing the language name to be JavaScript. The purpose of this name change was to connect the scripting language with the growing interest in Sun's Java programming language, however since then, this has created a substantial degree of confusion. Whilst there are many likenesses between JavaScript's and Java's core syntaxes, there are also significant disparities that exist between them in other areas.

Scripting Language

Programming languages that do not require assembly prior to execution are referred to as *interpreted* languages. JavaScript program is one such language. An *interpreter* is software that reads and executes a program that has been written in an interpreted language. A JavaScript interpreter is present in the majority of contemporary browsers.

Typically, there is a greater degree of difficulty in maintaining programs written in compiled languages such as Java then in managing programs written in interpreted languages. With the latter, there is no requirement to recompile interpreted programs following modification, and as there are no compiled versions of source files, there are less files overall that must be managed.

Furthermore, there are frequently less complications with programs written in an interpreted language than comparable programs written in a conventional compiled language. To illustrate this point, JavaScript has a single Number data type, whereas conversely, Java entails a broad range of numeric data types (i.e. int, float, double, etc.).

Data Types

As discussed earlier, in JavaScript, the variables do not have very explicit data types, however each one contains a value. There are six JavaScript data type categories as follows, into which each value can be classified as Numbers – for numeric values, String – for string values, Boolean – for the literals true and false values, Null – for the literal null values, Object – for object values and Undefined – for Variables that have been recognised but do not yet have a value.

In Java, the Java compiler flags the variables in the undefined category as errors, however in JavaScript, this is the responsibility of the developer. With the exception of the object category, the JavaScript data types are sometimes referred to collectively as *primitive data types*.

Operand Values	**typeof Returns**
Null	object
Boolean	boolean
Number	number
String	string
Object representing function	function
Object not representing function	object
Declared variable without value	undefined
Undeclared variable	undefined
Undeclared property of an Object	undefined

Table 2.1.1 Values returned by *typeof* for common operands

As presented in Table 2.1.1 [33], *typeof* is a JavaScript operator that gives detail about the value data type of a variable. The *typeof* operator is frequently employed to assess whether a variable has already been defined prior to use. It is important to note that

there are a number of reasons for the *typeof* operator marking a string undefined, rather than it simply being the case that it is the variable values' classification. The following is an example of JavaScript employing the *typeof* operator to test the data type,

```
var x;
var y;
y = "this is a string";
alert("x is " + (typeof x) + "\n" +      // output as undefined
      "y is " + (typeof y));             // output as string
```

The term of *identifiers* are the strings utilised to name variables and are case sensitive. In JavaScript, an acceptable identifier is any string starting with a letter or underscore, made up solely of these characters and digits, and is not a reserved word. Lastly, in a JavaScript program, in circumstances where a variable is assigned a value without first being declared with var statement, the variable will automatically be generated by the scripting engine.

Object Properties

In JavaScript, an *object* is defined as a set of *properties*, in which each one is comprised of an individual *name* and a value classified as one of the six aforementioned JavaScript data types. JavaScript properties are similar to Java instance variables, in that they are non-static variables which have been declared outside any method.

Object properties are similar to JavaScript variables in that they do not have data types, as only property values do so. In the example given below, which is syntactically valid in JavaScript, there is a sequence of statements that sequentially assigns Boolean, String, and Number values to an individual property *(prop)* of an object *(obj)* (which is taken to have been previously declared),

```
obj.prop = false;
obj.prop = "null";
obj.prop = 0;
```

Another significant disparity between Java and JavaScript programs is that the JavaScript does not define classes, although there are several aspects that are class-like. For instance, object constructors can be defined to generate objects and automatically determine their properties (this is discussed in more detail later in this section). Furthermore, JavaScript employs a *prototype* mechanism that gives a type of inheritance (this is not discussed in more detail as it is outside the scope of this paper). It is possible to add or remove properties and methods from a JavaScript object after its creation, however this is not the case with Java, where an objects' class defines its variables and methods. The lines below illustrate this point,

```
var obj1 = new Object();
obj1.prop = "test object";
delete obj1.prop;
```

The flexibility of JavaScript is also demonstrated by its dynamic property creation. However, a downside of this flexibility is that JavaScript is missing some security aspects. For instance, in an assignment statement, if a property name is mistyped, an error message will not be generated. Instead, an entirely new property will be produced. This is an important element to monitor in the process of debugging JavaScript code.

A new empty object is generated by a *new* expression, following which the specified constructor is called and provided with this new object, in addition to the specified values of the argument. At that point, the constructor can carry out initialisation on the object, which can entail the creating and initialising properties, adding methods to the object, and adding the object to an inheritance hierarchy where further properties and methods can be inherited.

With regards to the Object() constructor, the constructor does not add any properties or methods directly to the new object. However, the object is amended, thereby inheriting a number of genetic methods, such as default to String() and valueOf() methods, which are used when converting the object to both String and Number values. There is no

major benefit to the value generated by these default methods, but they do avert a runtime error when data type conversion is trying to be applied to the object.

An *object initialiser* is a useful shortcut tool in JavaScript that enables the creation of an empty object, and generating properties on this object to which values are assigned. It is called a new Object(). The following is an example of this,

```
var obj2 = { prop1:6, prop2:null, prop3:"some text" };
```

The object generated by the statement contains three properties, labelled prop1 (assigned the value 6), prop2 (assigned the value null), and prop3 (assigned the value "some text"). Subsequently, the variable *obj2* is assigned a reference to this object.

Object References

In JavaScript, the values of type Object are not actually objects; rather they are references to objects. Therefore, the JavaScript code below will generate an output of a "some text" string,

```
var obj1 = new Object();
obj1.data = "some";
var obj2 = obj1;
obj2.data += " text";
window.alert(obj1.data);
```

Comparably, an Object value is employed as an argument to a function or method, the object reference is delivered, and it is not a reproduction of the object itself. The JavaScript code below illustrates this point:

```
function objArgs(param1, param2) {
  param1.data = "changed";
  param2 = param1;
  window.alert("param1 is " + param1.data + "\n" +
               "param2 is " + param2.data);
return; }

var o1 = new Object();
o1.data = "original";
var o2 = new Object();
o2.data = "original";
```

```
objArgs(o1, o2);
window.alert("o1 is " + o1.data + "\n" +
             "o2 is " + o2.data);
```

The program generates two objects labelled object1 (o1) and object2 (o2), which are subsequently passed to the function objArgs() as arguments. Therefore, once the function is called, instantly param1 and param2 are copies of the object references in object1 and object2, respectively. This is shown in Figure 2.1.2(a) [33]. The data property of the object referenced by param1 and object1 is amended by the first statement of the function, as depicted in Figure 2.1.2(b). Next, the function amends param2 to be a replication of the object referenced in param1. An important point here is that this does not affect the variable object2 or the object referenced by this variable in any way, as shown in figure 2.1.2(c). As the result, param1, param2, and object1 will all present as "changed", while object2 shows "original".

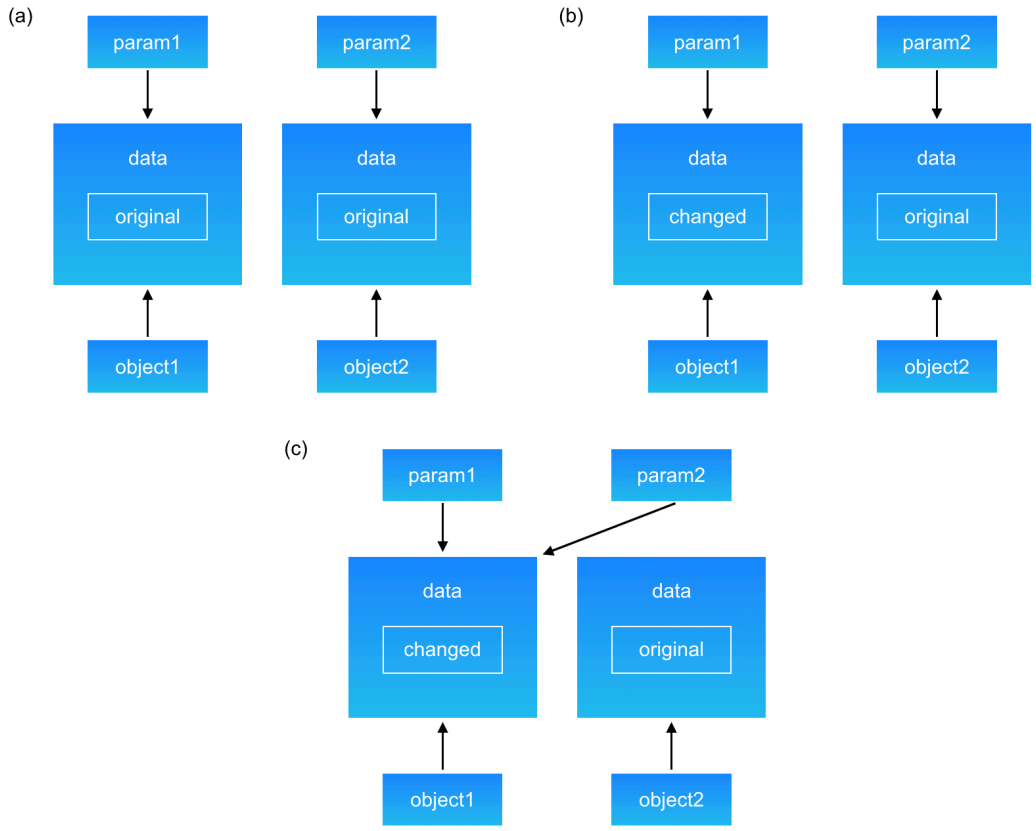


Figure 2.1.2 The state of variables and parameters of the object argument example (a) prior to the initial statement of the objArgs function is executed (b) following the execution of this statement (c) after the function's second statement is executed

2.1.3.3 Browser and DOM

A host object called document allows JavaScript programs to access the Document Object Model (DOM). Several other host objects are provided by frequently used browsers, however in contrast to document, these objects do not have any official standards. Hence, this section will provide a short overview of several common host objects, as well as discussing the document object. These objects have several functions, including enabling a JavaScript program to modify the size of browser windows, and move the browser to another URL.

The HTML specification includes intrinsic event attributes and the *meta* element, therefore nothing is directly related to the DOM at the moment. This will be first introduced in Figure 2.1.3 [33], which presents the JavaScript show() function. The function initially calls the getElementById() method of the document object, delivering the value of the first argument passed to it. This is the string *img1* in both calls to show(). To do so, a String is taken, and a JavaScript Object is returned, in which the *id* attribute value of the document element is the specified String. Subsequently, this object has a method setAttribute(), which permits JavaScript code to assign values to the attributes of an *img* element, for example, the *src* attribute. In order to assign the value of its second argument (a String representing a URL) to the *src* attribute of the *img* element, the show function() employs this method. Following this amendment, the browser will display a different image for this *img* element as shown,

```
function show(id, url) {
  var el = window.document.getElementById(id);
  el.setAttribute("src", url);
  return;
}
```

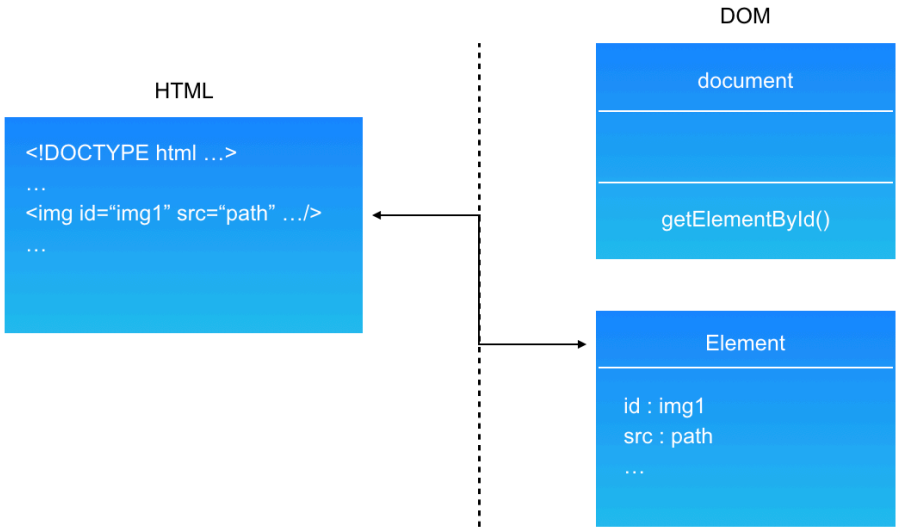


Figure 2.1.3 For each element in an HTML document there is a corresponding Object which can be accessed using getElementById, and the HTML attributes of an element can be specified using the setAttribute.

Intrinsic Event Handling

When something of potential interest takes place in a browser, this is referred to as an *event*. These could include occurrences when moving the mouse over an element, clicking the mouse button and pressing a key. Each of these is assigned an abbreviated name; for example, the first event listed above is a "mouseover". As observed earlier, an intrinsic event attribute is employed to give scripting code that is called at the point of occurrence of a specific event that is connected with the element. Lastly, the connected event follows the name given to each event attribute.

Table 2.1.2 provides the list of intrinsic event attributes as defined by the HTML 4.0 recommendation. They are applicable to the body element and to the vast majority of elements that are visually embodied in an XHTML 1.0 Strict document [33].

Attribute	**Events**
onload	The body of the document has been fully loaded and parsed by browser

onunload	A new document is ready to be loaded instead of the current one by browser
onclick	A mouse button has been clicked and released on the element
ondblclick	The mouse has been double-clicked on the element
onmousedown	The mouse has been pressed on the element
onmouseup	The mouse has been released on the element
onmouseover	The mouse has moved over the element
onfocus	The element has just received the focus
onblur	The element has just lost the focus
onkeypress	The key has been pressed and then realised on the element
onkeydown	The key has been pressed on the element
onkeyup	The key has been released on the element
onsubmit	The element is ready to be submitted
onreset	The element is ready to be reset
onselect	The element's text has been selected
onchange	The element's value has been changed

Table 2.1.2 The list of HTML intrinsic event attributes

Usually, a *meta* element is utilised to detail the information that would typically be located in the HTTP header field of the response message containing the HTML document: http-equiv is employed to denote the name and content value of the HTTP header field. Content-Script-Type is the name given to the HTTP header field defined by the meta element. It is utilised within the document to state the default language for scripts. It should be remembered that within web documents, whilst JavaScript is the most frequently used, other scripting languages are also effective when employed. So, the ultimate outcome is that the meta element informs the browser that the intrinsic event attributes are in JavaScript language.

```
<meta http-equiv="Content-Script-Type" content="text/javascript" />
```

(In the majority of browsers, if the content script type header field is undefined within a document, the default will be JavaScript.)

The Document Tree

For JavaScript programs that run within a DOM2-compliant browser, there are a range of node types comprising the accessible document tree. Some of these nodes are JavaScript objects that match HTML elements including *html* or *body,* whilst others may be comprised of text which signifies an elements' content or the white space between elements. Moreover, other nodes may be representative of HTML comments' text. The document type declaration is also represented by a node. Instances of a specific host object represent each form of node. For example, instances of the host object called Element represent document elements; likewise, instances of the host object named Text represent text and the white space between elements. The DOM determines a generic host object called Node that is comprised of properties that are also part of any of the document trees' objects, such as Element, Text and a number of other host objects. This is done to streamline the definition of these disparate host objects. Tables 2.1.3 and 2.1.4 provide lists of the core properties of the Node object [33].

Property	**Description**
nodeType	Represents the type of node in Number
nodeName	Provides the name of this Node in String
parentNode	References to the object of the node's parent
childNodes	Returns array that contains child nodes of this node
previousSibling	Returns previous sibling of this node, or null if not existed
nextSibling	Returns next sibling of this node, or null if not existed

attributes	Returns array that contains attributes of this node

Table 2.1.3 The list of non-method properties of Node object

Method	**Function**
hasAttributes()	Returns true if this node has attributes
hasChildNodes()	Returns true if this node has children
appendChild(Node)	Adds the Node to the end of the children list
insertBefore(Node, Node)	Adds the first Node to the children list
removeChild(Node)	Removes the Node from the children list
replaceChild(Node, Node)	Replace the second Node with the first in the children list

Table 2.1.4 The list of method properties of Node object

In detail, the documentElement property of the document object stores the Element instance that represents the *html* element of the document. By beginning at this node, it is a simple process to traverse the document tree using the Node methods, thereby ascertaining the required information about the tree.

Value	**Constant**	**Type**
1	Node.ELEMENT_NODE	Element
2	Node.ATTRIBUTE_NODE	Attr
3	Node.TEXT_NODE	Text
4	Node.COMMENT_NODE	Comment
5	Node.DOCUMENT_NODE	Document
6	Node. DOCUMENT_TYPE_NODE	DocumentType

Table 2.1.5 The list of possible values for the *nodeType* property of Node object

Although the root of a HTML document is the *html* element, it is actually in the DOM that the document object is considered the Node trees' root. It is the HTML Element instances' parent, whereas the value of its own parentNode is null. Therefore, external to the *html* element, document provides a parent for nodes representing mark up. As illustrated in Table 2.1.5, this includes comments and the document type declaration (which has a nodeType value of Node.DOCUMENT_TYPE_NODE).

Furthermore, ELEMENT_NODE type nodes are instances of the element host object. Similar to document, they have particular extra properties in addition to those belonging to Node. TagName is the only non-method property that is specific to Element, and fundamentally, it is simply another name for the nodeName property. Character data is represented by instances of the TextDOM object. This basically refers to anything that is not markup in the HTML document. Node.TEXT_NODE is the node type for these elements. Data is the key property of text instances and is represented by the Text node. Text content that is presented by the browser is modified by assigning a value to this property.

DOM Event Handling

When an event takes place in the DOM event model, an instance of a host object is created, which is named Event. This instance is comprised of event information which details the type of event, for example, click or mouseover that were previously mentioned, and a reference to the document node that matches the markup element that caused the event. This node is referred to as the *event target*. This information is given by the event instance properties type and target.

An event instance is deployed to specific event listeners as soon as it is generated. In the JavaScript form of the DOM, an *event listener* is defined as an operation that selects a single argument from the instance of event. An association between an event listener

and the type of event taking place on a node object is generated by a call to the addEventListener() method on that node. For instance, in the case of a document encompassing an element with a msgBtn id, the JavaScript code below is executed,

```
var btn = window.document.getElementById("msgBtn");
btn.addEventListener("click", msgBox, false);

function msgBox(el) {
  window.alert(
    "Some Text\n" +
    "Event type: " + el.type + "\n" +
    "Event target element type: " + el.target.nodeName);
return; }
```

In the later section, there will be further discussion of the third (Boolean) argument to addEventListener() in association with event capture, however for the moment, this is set to false. The second argument relates to the event listeners identifier. The first argument is a String detailing the type of event listened for, which is not case sensitive. Most of the event types are the same for the DOM2 and those employed for the HTML intrinsic event attributes, but the following three intrinsic event types do not have a corresponding DOM2 event type: keypress, keydown and keyup. In addition, there is no DOM2 double-click event, which is one that matches the ondblclick attribute.

Event Propagation

Besides defining event instances, the DOM2 event model also manages *event propagation*. Within the document tree, the target node of a mouse event is the most deeply nested screen visible node that covers the mouse location. For instance, the anchor node is the target of the mouseover event when the following criteria are met: (1) the mouse travels over an anchor (i.e. a hyperlink) within a paragraph element, that (2) itself is in a td element, that (3) is nested within a table elements hierarchy, which (4) itself is part of the document body element.

Whilst an event has a single target node, it can result in the calling of numerous event listeners. This can initially transpire if a number of event listeners have been added to the target node. To be more specific, when an event takes place, a DOM2-compliant browser theoretically generates a list of event listeners, which are then called based on the list order. This list is comprised of three types of event listener, which, in order, are: capturing listeners, target listeners, and bubbling listeners. A *capturing event listener* is connected with a document tree ancestor of the target node and was generated via a call to addEventListener() with a true third argument. A *target listener* is directly added to the target node with a false third argument. Finally, a *bubbling listener* is connected with a document tree ancestor of the target node and was generated via a call to addEventListener() with a false third argument.

In the case that there are numerous ancestors with capturing listeners, their order runs from those closest to the document tree root, moving to those nearest the target node. The opposite occurs for bubbling listeners, as the order begins with nodes closest to the target and concludes with those nearest to the root. For some event types such as load, unload, focus, and blur, bubbling listeners are excluded from the listener list.

Following the creation of this list of listeners, the browser calls them individually as per the order of the list. There are two previously unmentioned DOM2 properties of event which inform the listener function about event propagation. The first property, eventPhase, contains a Number value that denotes the browsers' *event processing phase*. A number value of 1 signifies that the call is to capturing event listener, which means that the browser is in the capture phase. A number value of 2 indicates a target listener is being called; whilst a number value of 3 represents a bubbling listener. The second property, currentTarget, encompasses a reference to the listeners' registered node.

It is possible but the browser will not call all of the listeners from the ordered list if any one of the listeners calls the stopPropagation() method on its Event instance argument,

as it takes no arguments. Specifically, as soon as stopPropagation() is called, any listeners that would be called during the present processing phase and that are registered listeners on the current node (currentTarget) will be executed; however, at that point, there will be no additional listener processing for this event. It is important to note that a call to stopPropagation() in an event listener for one event does not affect any other events.

Capturing event listeners are especially effective for tasks that are related to the top levels of the document tree. For instance, this would apply to the cursor trail example that was given earlier in this document. Typically, it is preferable that tasks such as these are carried out on every event possible. When an event is captured prior to reaching listeners at lower levels of the document tree, it means that the capturing listener can process the event regardless of whether stopPropagation() is called by a listener further down the ordered list.

Generating Events

When invalid data is entered into a text box, it is preferable that firstly, the user is informed that there is an error, and secondly, that the contents of the text box are selected. The purpose of this is to both highlight the information and to automatically replace the contents when new valid data is entered. Ideally, the browser it would modify its display in the same way it changes when the contents of the textbox are selected with the mouse by the user. This would also cause the occurrence of a select event.

Method	**Elements**
blur	anchor, input, select, textarea
click	input (button, checkbox, radio, submit)

focus	anchor, input, select, textarea
select	input (text, file), textarea

Table 2.1.6 The list of DOM2 methods for generating common events

As shown in Table 2.1.6, DOM2 establishes four methods that can be employed to simulate effects such as these. These methods take no arguments and have no return value. Each one creates an event, the type of which is determined by the name of the method, and the object becomes the Events' target value. Moreover, the browser displays any visual adjustments that are usually connected with the event, including selecting the text within a textbox. Whilst these four methods are adequate for most typical requirements, they are actually special cases that are components of a more general DOM mechanism. This mechanism can be employed to create random events.

2.1.4 Data-Driven Documents

2.1.4.1 Selections and Data

Selections enable the various data-driven transformation of the Document Object Model (DOM) including set attributes, styles, properties, HTML or text content [36]. Elements can be added or removed to match the data by using the data joins' enter and exit selections.

Usually, selection methods return the current or a new selection, allowing the brief application of multiple operations through method chaining on a specific selection. For instance, the following demonstrates appointing the class and colour style of all aspects of a paragraph in the specified document,

```
d3.selectAll("p")
    .attr("class", "main")
    .style("color", "green");
```

which is equal to,

```
var p = d3.selectAll("p");
p.attr("class", "main");
p.style("color", "green");
```

All selections are immutable, and a new selection is returned by every selection method that impacts which elements are selected, rather than the existing selection being amended. However, a significant point is that elements are necessarily changeable as the document transformations are selection-driven.

Element Selection

W3C selector strings are accepted by selection methods. These include ".box", which selects elements with the class box, or "div", which selects the DIV elements. There are two forms of selection methods: firstly, select, and secondly, selectAll select only chooses the first matching element whereas selectAll chooses all matching elements in document order. The top-level selection methods query the whole document (d3.select

& d3.selectAll), while the sub-selection methods limit selection to the descendants of the selected elements (selection.select & selection.selectAll).

Specifically, d3.select selects the first element that corresponds to the specified selector string. An empty selection is returned if there are no corresponding elements. If several elements correspond to the selector, the first matching element per the document order will be selected alone. For instance, to select the first DIV element,

```
var div = d3.select("div");
```

The specified node is selected instead in circumstances where the selector is not a string. This is beneficial if the node reference is already known, as is the case within an event listener or a global such as the document body. The following demonstrates making a clicked DIV element green,

```
d3.selectAll("div").on("click", function() {
  d3.select(this).style("color", "green");
});
```

d3.selectAll selects all elements that correspond to the selector string specified. The elements are selected based on the document order, running from top to bottom. An empty selection is returned if there are no elements that correspond to the selector in the document or if the selector is null or undefined. The following demonstrates selecting all DIV elements,

```
var divs = d3.selectAll("div");
```

The array of nodes specified are selected when the selector is not a string. This is beneficial if the node reference is already known, as with the this.child.Nodes within an event listener, or a global such as document.links. As an alternative, the nodes may be a pseudo-array such as a NodeList or arguments. The following demonstrates making all links green,

```
d3.selectAll(document.links).style("color", "green");
```

Binding Data

The elements are successfully bound to the data when the specified array of data binds with the selected elements and a new selection representing the update selection is returned [36]. In addition, the enter and exit selections are defined on the returned selection, and they can subsequently be employed to add or remove elements that match the new data. The specified data is an array of random values such as numbers or objects, or a function that returns an array of values for each group. The data assigned to an element is stored in the property __data__, thereby causing it to be "sticky" data that is available on reselection.

Each group in the selection has specified data. In general. the data should be specified as a function if the selection has multiple groups, which include d3.selectAll followed by selection.selectAll. This function will be assessed for each group in order and is subsequently passed the following components: the parent datum of the group *d*, which could be undefined, the group index *i*, and the parent node of the selection *nodes*, which serves as the parent element of the group. Selection.data can be utilised and combined with selection.join to enter, update, and exit elements to correspond to data, and explicitly selection.join is comprised of selection.enter, selection.exit, selection.append, and selection.remove.

In circumstances where the key function is not specified, the first datum in data is assigned to the first selected element, followed by the second datum to the second selected element, and so forth. A key function can be specified to control the data-element assignment. This substitutes the default join-by-index, as it computes a string identifier for each data element.

In detail, the update and enter selections are returned in data order, and before the join, the exit selection preserves the selection order. When the key function is specified, the element order in the selection may not correspond to their document order. In this case, selection.order or selection.sort can be applied where necessary. If the data is not

specified, this approach means that the array of data for the selected elements will be returned.

Fetching Data

Typically, data is formatted in a standardised format file, irrespective of the source of data. Some of the more frequently used formats are XML, CSV, and JSON [37]. D3 has made data fetching functions available to support these data which has proved to be useful. With regards to formats, JSON and XML can incorporate the nested structure of the data; however, CSV is not capable of supporting this.

D3 also offers handy integral parsing on top of the fetch module. The following demonstrates how to load a text file,

```
d3.text("/text-file.txt").then(function(data) {
  console.log(data); // output text from the file
});
```

And to load and parse a CSV file,

```
d3.csv("/csv-file.csv").then(function(data) {
  console.log(data); // output data in CSV format
});
```

A standard D3 has five integral data fetching functions which support the main data formats. These include d3.text(), d3.xml(), d3.html (), d3.csv() and d3.json(). Of these five, d3.json() and d3.csv() are the two most often used functions.

2.1.4.2 Scales and Axes

Scales are a helpful construct for a fundamental visualisation task of mapping an aspect of abstract data to a visual representation [38]. Whilst scales are most frequently utilised for position encoding quantitative data (including mapping a measurement in metres to a position in pixels for dots in a scatter plot), scales can denote almost any visual encoding, including deviating colours, stroke widths, or symbol size. Furthermore, scales and almost any type of data are compatible, including named categorical data or discrete data that needs reasonable breaks.

Normally, a linear scale is most appropriate for continuous quantitative data. In contrast, a time scale is best suited to time series data. Based on the requirements of the distribution, transforming data utilising a power or large scale could be an optional avenue. A quantise scale can support differentiation by rounding the continuous data to a fixed set of discrete values. In the same manner, a quantile scale calculates quantile from a sample population, and a threshold scale facilitates the specification of random breaks in continuous data.

With regards to discrete ordinal or categorical data, which are ordered and unordered data respectively, an ordinal scale specifies a definitive mapping from a set of data values to a matching set of visual attributes; for example, colours. The associated band and point scales are helpful for position-encoding ordinal data. This could refer to bars in a bar chart or dots in a categorical scatter plot. Scales do not have an inherent visual representation, however most of them can produce and format ticks for reference marks, which is beneficial to the construction of axes.

Continuous Scales

The function of continuous scales is to mark a continuous quantitative input domain to a continuous output range. The mapping maybe inverted if the range is also numeric. A

continuous scale is indirectly constructed; therefore, a linear, power, log, identity, time, or sequential colour scale could be utilised in its place.

A specified value from the domain returns a matching value from the range. If circumstances arise that a given value is outside the domain and clamping is not possible, the mapping can be extrapolated to the point that the returned value is outside the range. Here is an example of how to apply a position encoding,

```
var x = d3.scaleLinear()
    .domain([0, 100])
    .range([0, 800]);

x(10); // 80
x(30); // 240
```

Sequential Scales

There are similarities between continuous scales and sequential scales, like diverging scales. Both map a continuous numeric input domain to a continuous output range. The output range of a sequential scale is determined by its interpolator and is not configurable, which differentiates it from a continuous scale. Invert, range, rangeRound, and interpolate methods are not exposed by these scales.

d3.scaleSequential produces a new sequential scale with a given domain and interpolator function. An unspecified domain defaults to [0, 1], whilst an unspecified internal interpolator defaults to the identity function. Upon application of the scale, the interpolator will be called upon with the value that is normally in the range [0, 1], where 0 and 1 are the minimum and maximum values respectively. The implementation of the prevalent rainbow scale with d3.interpolateRainbow is shown as,

```
var rainbowScale = d3.scaleSequential(d3.interpolateRainbow);
```

Diverging Scales

There are similarities between diverging scales, like sequential scales, and continuous scales. Both map a continuous numeric input domain to a continuous output range. However, in contrast to continuous scales, a diverging scales' output range is determined by its interpolator and is not configurable. Invert, range, rangeRound, and interpolate methods are not exposed by these scales.

d3.scaleDiverging builds a new diverging scale within a given domain and interpolator function. An unspecified domain defaults to [0, 1]; and an unspecified interpolator defaults to the identity function. Upon application of the scale, the interpolator will be instigated with the value normally in the range [0, 1]. Here, 0, 0.5, and 1 represent extreme negative, neutral, and extreme positive values, respectively. The following is an example of how to implement a diverging spectral scale,

```
var spectral = d3.scaleDiverging(d3.interpolateSpectral);
```

Ordinal Scales

Ordinal scales differ from continuous scales in that they contain a discrete domain and range. d3.scaleOrdinal generates a new ordinal scale with a given domain and range. Both an unspecified domain and an unspecified range will default to the empty array. Until the point that a non-empty range is determined, an ordinal scale will always return undefined.

Band Scales

Aside from their output range being continuous and numeric, band scales are similar to ordinal scales. The band scale automatically computes discrete output values by dividing the continuous range into homogeneous bands. Normally, band scales are utilised for bar charts that have an ordinal or categorical dimension. A band scale has

an unknown value; hence, it is essentially undefined. Additionally, band scales do not permit indirect construction of domains. Figure 2.1.4 presents an example of the band scale [38].

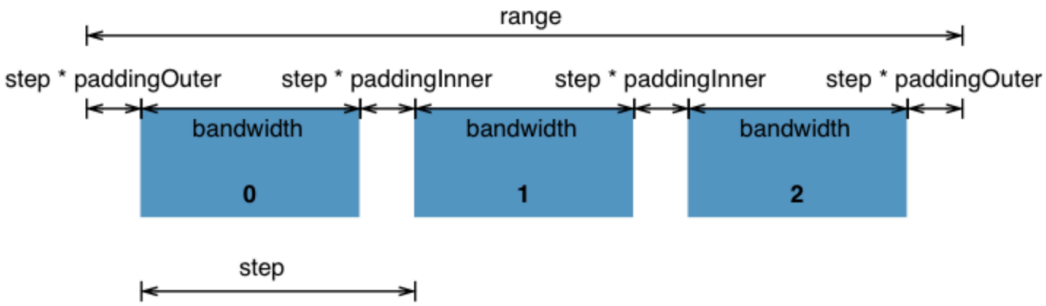


Figure 2.1.4 An illustration of the band scale which can be created by d3.scaleBand

Within a specified domain and range, and without padding, rounding, or centre alignment, the d3.scaleBand constructs a new band scale. An unspecified domain defaults to the empty domain, whilst an unspecified range defaults to the unit range [0, 1].

Point Scales

Point scales are a type of band scale where the bandwidth is set at zero. Usually, point scales are utilised for scatter plots that have an ordinal or categorical dimension. The value of a point scale is unknown, so it is always undefined; therefore, a point scale does not permit indirect construction of a domain. Figure 2.1.5 presents an example of the point scale [38].

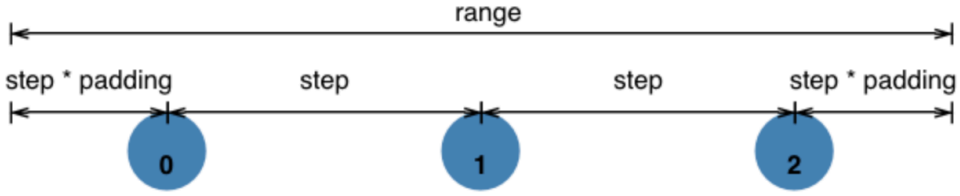


Figure 2.1.5 An illustration of the point scale which can be created by d3.scalePoint

Within the given domain and range, and without padding, rounding, or centre alignment, the d3.scalePoint builds a new point scale. An unspecified domain defaults to the empty domain, whilst an unspecified range defaults to the unit range [0, 1].

The Axes

The axis element provides reference marks for scales that can be read by the human users. Axes are always rendered at their origin, regardless of their orientation [39]. In terms of chart position, to change the axis, the transform attribute on the containing element must be specified. For instance,

```
d3.select("body").append("svg")
    .attr("width", 600)
    .attr("height", 30)
  .append("g")
    .attr("transform", "translate(0,30)")
    .call(axis);
```

The elements comprising an axis are firstly a path element of class “domain”, which represents the degree of the scales’ domain; and secondly, transformed g elements of class “tick”, which denotes each of the scales’ ticks. Each one contains a line element to facilitate drawing the tick line, and a text element that is used for the tick label. Figure 2.1.6 provides an illustration of the D3 generated axes in horizontal and vertical [40].

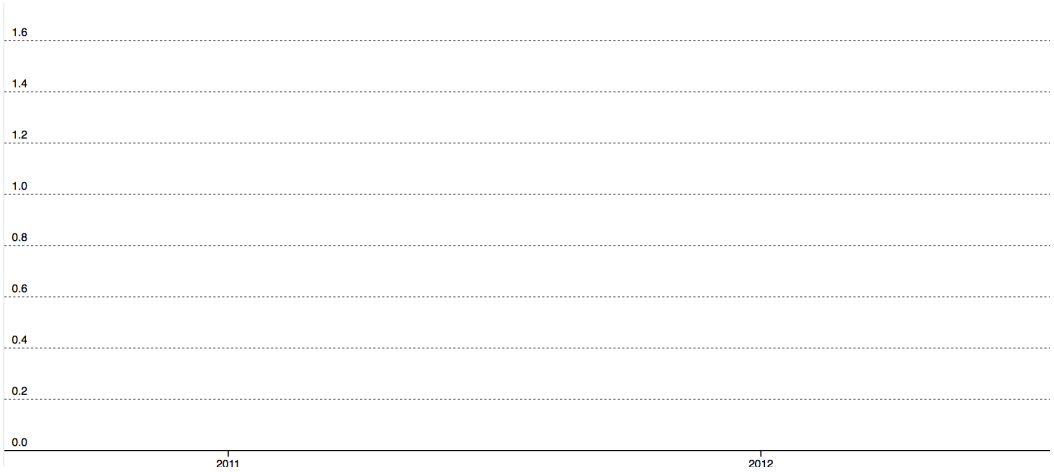


Figure 2.1.6 An illustration of using D3 axes to create horizontal and vertical axes

2.1.4.3 Transition and Timer

D3 transition is a selection-like interface that is used to animate amendments to the DOM [41]. Rather than immediately applying modifications, the transitions gradually interpolate the DOM over a specified period, from the present to the chosen target states. The stages of applying a transition are firstly, to select elements, secondly, to call select.transition, and thirdly, to make the necessary amendments. For instance,

```
d3.select("div")
  .transition()
    .style("height", "600px");
```

The majority of selection methods are supported by transitions, which include transition.attr and transition.style in place of selection.attr and selection.style. However, not all methods are supported; for instance, prior to beginning a transition, elements must be appended, or data must be bound. In order to simplify the removal of elements following the end of a transition, a transition.remove operator is provided.

Transitions leverage a range of integral interpolators for the purpose of computing the intermediate state. There is automatic detection of colours, numbers, and transforms. Additionally, strings with embedded numbers are identified, which is the case with many styles, including padding and font sizes, and paths. transition.attrTween, transition.styleTween or transition.tween can be employed to specify a custom interpolator.

Transition Selection

Transitions originate from selections through selection.transition, and d3.transition can be utilised to generate a transition on the document root element. To be more specific, selection.transition returns a new transition on the specified selection with the given name. Null is returned if a name is not specified. Transitions must have the same name for them to be exclusive with another.

In circumstances, where the name is a transition instance, the returned and specified transitions will have identical IDs and names. The existing transition is returned for an element if there is a pre-existing transition with the same ID on the given element. if not the timing of the returned transition is according to the inheritance from the existing transition of the same ID on the closest ancestor of each specified element. Therefore, this method is effective for synchronising a transition across multiple selections, or re-selecting a transition for specific elements and amending its configuration. For instance,

```
var t = d3.transition().duration(900);

d3.selectAll(".leaves").transition(t)
    .style("fill", "green");

d3.selectAll(".flower").transition(t)
    .style("fill", "red");
```

The default timing parameters are employed when a specified transition is not identified on a selected node or its ancestors, which would indicate that the transition had concluded.

In addition, selection.interrupt suspends the specified name's active transmission on selected elements, and terminates any upcoming transitions with the specified name. Null is used if a name is not specified.

If a transition element is interrupted, there are no repercussions for any transitions on any descendant elements. For instance, an axis transition is comprised of numerous independent synchronised transitions on the descendants of the axis element, which refers to the tick lines, the tick labels, the domain path, and so forth. Therefore, the descendants must be interrupted in order to interrupt the axis transition.

```
selection.selectAll("*").interrupt();
```

Transition Lifecycle

As soon as a transition is generated by selection.transition or transition.transition, the transition can be configured utilising approaches such as transition.delay, transition.duration, transition.attr, and transition.style. Methods with specified target values such as transition.attr are assessed concurrently; however, other methods that need the starting value for interpolation such as transition.attrTween and transition.styleTween must be postponed until the transition has started.

The transition is scheduled soon after creation, either at the end of the current frame or during the subsequent frame. When this happens, it is no longer possible to amend the delay and start event listeners.

When the next transition starts, if there is an active transition which has the same name on the same element, it is interrupted. Then, an interrupt event is passed to the registered listeners. Note that the interruption occurs at the start rather than at the creation; and therefore, even a transition that has no delay will not immediately interrupt the active transition, because the old transition is provided with a final frame. If it is necessary to interrupt immediately, selection.interrupt can be utilised. In addition, the starting transition terminates any pending transactions of the same name on the same element that had been generated prior to start the transition. Subsequently, the transition passes a start event to the registered listeners. Once this point has passed, the transition can no longer be amended, as when it is running, the transition's timing, tweens, and listeners cannot be adjusted.

The transition begins during the frame, but it is after all transitions starting this frame have started that the transition initially calls upon its tweens. Usually, batching tween initialisation entails reading from the DOM, and its advantage is that it enhances performance by circumventing interleaved DOM reads and writes.

Each frame of an active transition initiates its tweens with an eased t-value that ranges from 0 to 1. The tweens are invoked by the transition as per their registration order within each frame.

A transition invokes its tweens for the last time with a non-eased t-value of 1 when it is ending. It subsequently deliveries an event to registered listeners. This is the final point at which the transition can be examined. Upon its conclusion, the transition is deleted from the element and its configuration is terminated.

The Timer

A D3 timer offers an efficient queue that can handle thousands of animations simultaneously, whilst ensuring consistent synchronised timing with animations that are concurrent or staged. Internally, if it is available, it employs a requestAnimationFrame for fluid animation. In the case of delays lasting longer than 24ms, it can be switched to setTimeout [42].

Specifically, d3.now returns the current time according to the performance.now, if available, and the date.now if not. At the beginning of each frame, the current time is updated, thereby ensuring its consistency during the frame. Also, any timers that have been scheduled during the same frame will therefore be synchronised. In the event that this method is called outside of a frame, the current time is calculated and fixed until the next frame, which also safeguards the consistency of timing during the event management.

A new timer is scheduled by the D3 timer, which subsequently calls upon the given callback back continually, until the timer is stopped. The user can choose to add a numeric delay in milliseconds to invoke the specified callback after a delay. If the delay is unspecified, it will default to 0. The delay is associated with the specified time in milliseconds; therefore, if the time is unspecified it will default to now.

The apparent elapsed time since the timer had an active status is passed to the callback. For instance,

```
var timer = d3.timer(function(t) {
  console.log(t);
  if (t > 200)
    timer.stop();
}, 50);
```

The following console output can be expected:

```
3
25
..
209
```

The initial elapsed time *t* is 3ms. This refers to the elapsed time from the commencement of the timer, not from when the timer was scheduled. The timer, in fact, started 50ms after the scheduled time, because of the delay that had been specified. However, the apparent elapsed time may actually be shorter than the real (true) elapsed time if it transpired that the page is backgrounded and requestAnimationFrame is paused, because this means that the apparent time is frozen in the background.

If it occurs that the timer is called during the callback of another timer, the new timer callback will be initiated immediately at the end of the current frame, instead of waiting for the next frame. This depends on whether the new timer callback is eligible according to the criteria of the specified delay and time. Within the frame, it is a certainty that the timer callbacks will be invoked in the order that they were scheduled, irrespective of their start time.

2.1.4.4 Shapes and Colours

In general, visualisations are comprised of discrete graphical marks, including symbols, arcs, lines, and areas. Certain shapes such as rounded annular sectors and centripetal splines are complicated, whereas the rectangles of a bar chart are relatively simple to generate directly using SVG or Canvas. D3 shape provides a range of shape generators for the convenience of the developers [43].

Like other components of D3, these shapes are data-driven, in that each shape generator reveals accessors that determine how the input data is mapped to a visual representation. For instance, scaling fields of data to fit a chart can expose a line generator for a time series,

```
var lineGen = d3.line()
    .x(function(d) { return x(d.x); })
    .y(function(d) { return y(d.y); });
```

This line generator can subsequently be utilised to calculate the d attribute of an SVG path element,

```
path.datum(data).attr("d", lineGen);
```

Arc

Like those shapes in a pie or donut chart, the arc generator creates a circular or annular sector. It would create a complex a complete circle or annulus if there is a difference of greater than τ between the beginning and end angles, which is the angular span. In contrast, if the angular span is less than τ, the arcs could have angular padding and rounded corners. Arcs are constantly countered at ⟨0,0⟩; hence transform is employed to transfer the arc to another position.

Pie

A shape is not directly produced by the pie generator; instead the pie generator calculates the necessary angles to show a tabular dataset in a pie or donut chart form. Subsequently, these angles can be passed to an arc generator. Figure 2.1.7 presents an illustration of the pie with an arc generator.

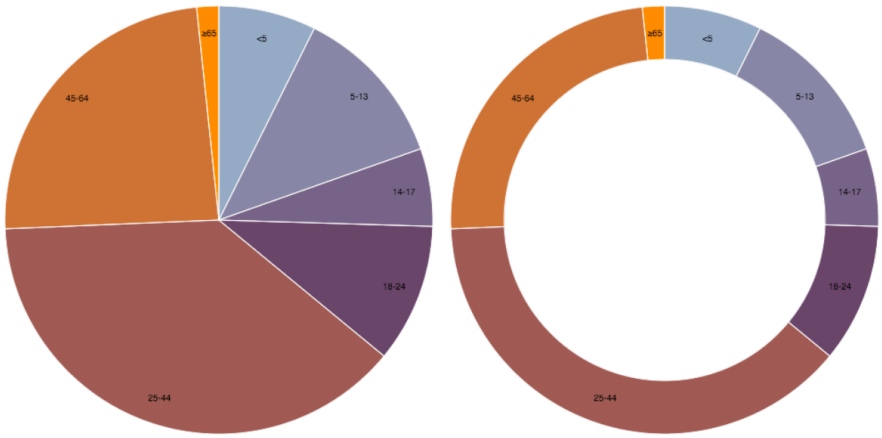


Figure 2.1.7 An illustration of using d3.pie with arc to create pie and donut charts

A new pie generator with the default settings is constructed by d3.pie. It produces a pie for the current array of data, returning an array of objects that signify the arc angles of each item of information. Any further arguments are random, and they are merely distributed to the accessor functions of the pie generator with *this* object. The length of the return array and the data are identical, and each element *i* in the return array has a matching element *i* in the input data. The following properties are resent in all the objects in the returned range,

- data - the input data for arc
- value - the arc's numeric value
- index - the arc's zero-based sorted index
- startAngle - the arc's start angle
- endAngle - the arc's end angle
- padAngle - the arc's pad angle

This representation is intended to complement the arc generators default start.angle, end.angle, and pad.angle accessors. Whilst the angular units are random, if the pie generator is going to be used with an arc generator, it is important that the angles are specified in radians.

Line

As with a line chart, the line generator generates a spline or polyline. Lines will also be present in numerous other visualisation types, including the links in hierarchical edge bundling. Figure 2.1.8 presents an example of a D3 generated line [44].

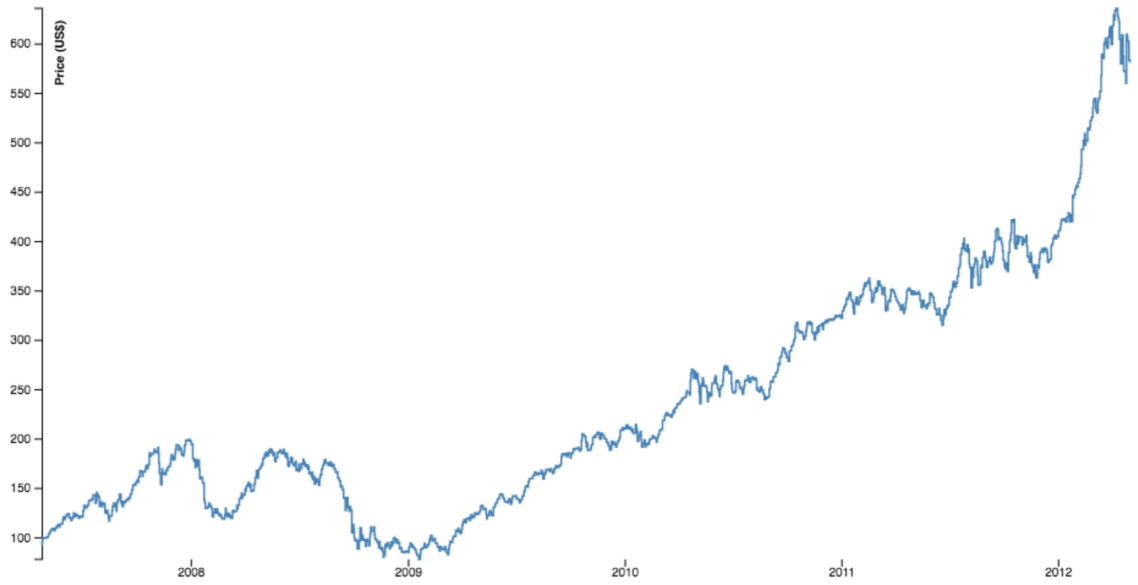


Figure 2.1.8 An illustration of using d3.line to create a line chart

d3.line produces a line for the selected data array. According to the associated curve of this line generator, the selected input data may require sorting by x-value prior to being passed to the line generator. The line is rendered to the context of the line generator (if present) as a sequence of path method calls, and a void result is returned with this function. If not, a path data string is returned.

Area

As with an area chart, the area generator generates an area which is delineated by 2 bounding lines that are either splines or polylines. Usually, both lines have identical x-values (x0 = x1), however they have different y-values (y0 and y1). In most cases, y0 is stated as a constant representing zero. The first line, which is the top-line, is defined by x1 and y1, and is firstly rendered. The second line, which is the baseline, is defined by x0 and y0, and is second rendered. Here, the points are in reverse order. Figure 2.1.6 provides an example of a D3 generated area chart .

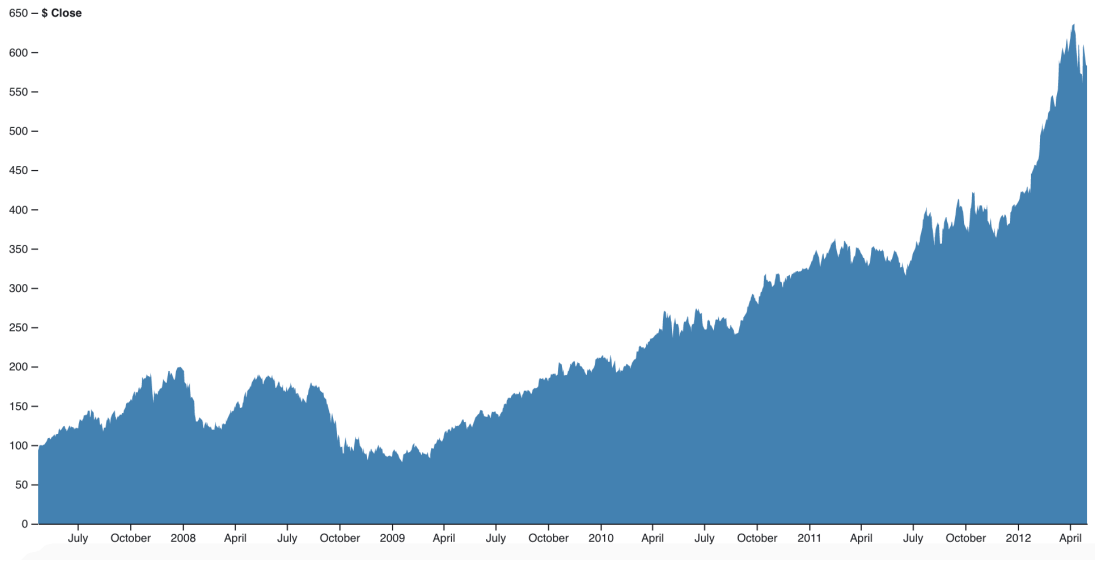


Figure 2.1.9 An example of using d3.area to create the area chart

d3.area produces an area for the selected data array. According to the associated curve of this area generator, it may be necessary to sort the selected input data by x-value prior to being passed to the area generator. The area is rendered to the context of the area generator (if present) as a sequence of path method calls, and a void result is returned with this function. If not, a path data string is returned.

Curves

As discussed, lines are sequences of two-dimensional [x, y] points, and areas are delineated by a top line and a baseline. However, the question remains as how to transform this discrete representation into a continuous shape; namely, how to interpolate between the points. A range of curves are given to show this. Figure 2.1.10 presents an example of a D3 generated curve.

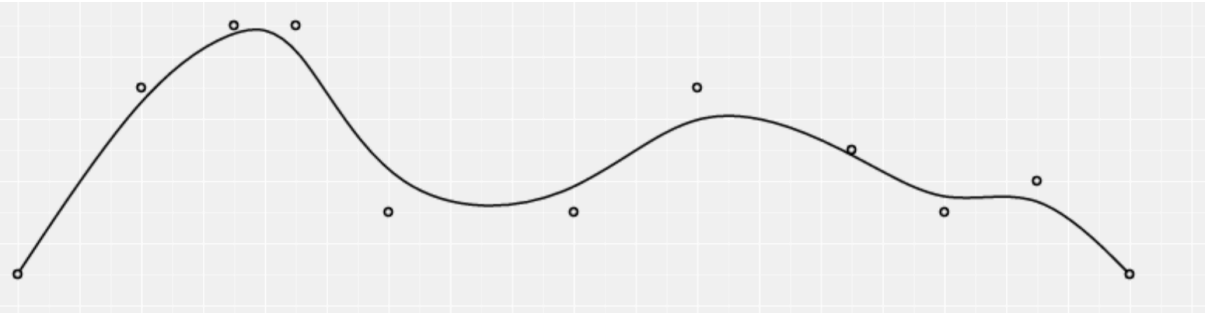


Figure 2.1.10 An example of using D3 curve to create a curve illustration

d3.curveBasis generates a cubic basis spline by utilising the specified control points. The first and final points are triplicated. This is done so that the spline begins and ends at the first and last points respectively, and is tangent to the line between the first and second points, and to the line between the second-last and last points.

Stacks

Stacking is possible for certain types of shapes, and is done by placing one shape alongside another. For instance, considering a monthly sales bar chart; it could be broken down by product category into a multi-series bar chart, in which the bars are stacked vertically. This is akin to subdividing a bar chart via an ordinal dimension, for example, a product category, and subsequently applying coloured encoding.

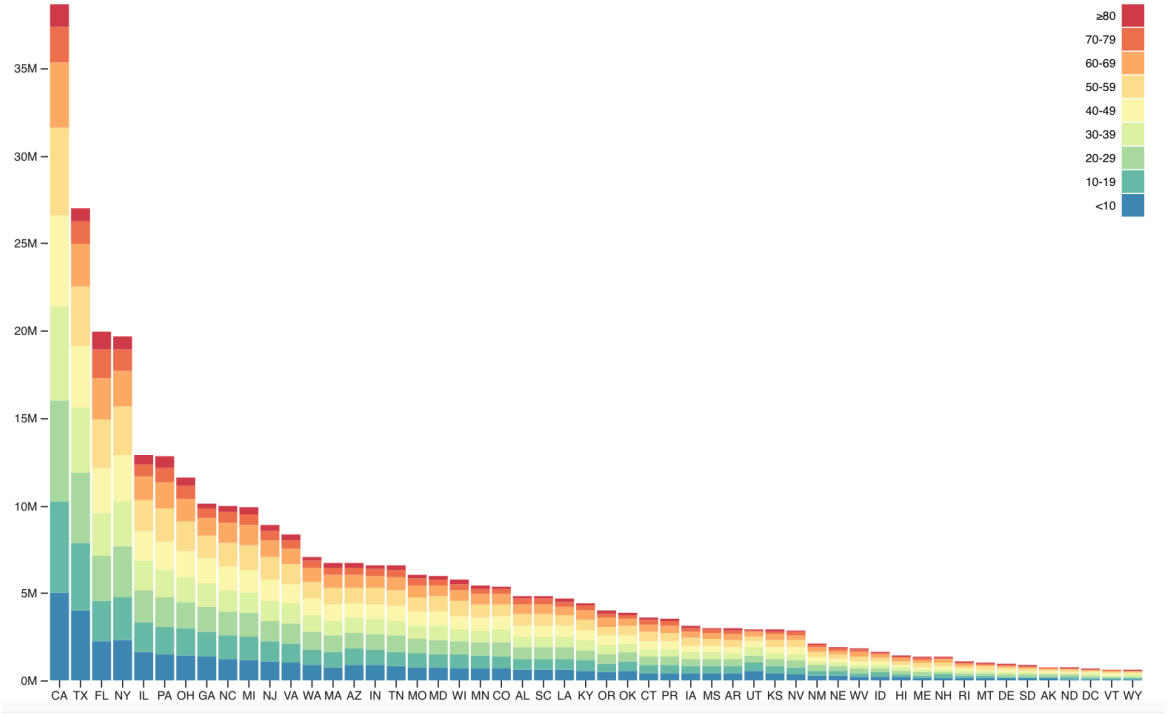


Figure 2.1.11 An example of using d3.stack to create the stacked chart

Stacked charts can illustrate the overall value and the per category value at the same time. However, doing so makes comparison across categories more complicated, as with the exception of the bottom layer, the layers of the stack are not aligned. Figure 2.1.11 presents an illustration of a generated stacked chart [45].

In the case with the pie generator, the stack generator does not directly generate a shape. Instead, it calculates the positions at which passing to an area generator or direct employment is possible; for instance, to position bars. To be more specific, the d3.stack generates a stack for the data array at hand, returning an array that represents each series. Any further arguments are random, and they are merely distributed to accessors alongside *this* object.

The Colours

Typically, browsers have an in-depth understanding of colours, however with JavaScript, there is only little guidance about manipulating them. Hence, D3 colour gives representations for a range of colour spaces, and facilitates colour specification, conversion, and manipulation [46]. For instance, considering the colour named "forestgreen",

```
var colour = d3.color("forestgreen");
```

If converting it to HSL,

```
var colour = d3.hsl("forestgreen");
```

Now the hue can be rotated by 180°, the saturation can be added, and it can be formatted as a string for CSS,

```
colour.h += 180;
colour.s += 0.3;
var colourStr = colour + "";
```

D3-color supports the abundant and machine-friendly RGB and HSL colour space, as well as other less common colour spaces [46].

Specifically, d3.color parses the selected CSS Colour Module Level 3 specifier string and an RGB or HSL colour is returned. A null value is returned if the specifier was invalid. CSS details the list of supported named colours. A noteworthy point is that the function can also be employed with *instanceof* in order to test if an object is a colour instance. This is also the case with colour subclasses, where the user can examine whether a colour is in a specified colour space.

2.1.4.5 Hierarchies and Geographies

By the nature, many data sets are inherently hierarchical, such as the geographic aspects including census blocks, census tracts, counties and states; the command structures in businesses and governments, as well as the software packages and file systems. Even data which is not hierarchical in nature can actually be hierarchically organised via approaches such as phylogenetic trees or k-means clustering.

Regarding D3 hierarchy, there are a number of frequently employed techniques to visualise hierarchical data as follows [47],

- *Node-link diagrams*: which display topology by utilising discrete marks for nodes and links. For example, a circle represents each node and a line links each parent and child. The neatness of the 'tidy' tree is very appealing, and the dendrogram sets leaves at the same level, in both polar and cartesian styles. The ease of interactive browsing is increased with the use of indented trees.
- *Adjacency diagrams*: which display topology via the comparative nodes' placement. For each node area, they can also encode a measurable facet. There are several purposes for this, including displaying the revenue or file size. Rectangles and annular segments are utilised in the 'icicle' and 'sunburst' diagrams, respectively.
- *Enclosure diagrams*: which also have an encoded area; however, they display topology through containment. The area is subdivided into rectangles repetitively by a tree map, and the circles are nestled compactly via circle-packing. Whilst it arguably more easily displays the topology, it is less efficient in terms of space than a tree map.

An effective hierarchical visualisation enables rapid multiscale inference. This entails micro-observations of single aspects, and macro-observations of large groups.

Cluster

The cluster layout can generate dendrograms, which are node-link diagrams that set the trees' leaf nodes at identical depths. Generally, dendrograms are less condensed than tidy trees; however, they are beneficial when it is necessary to place all leaves at the same level, as in phylogenetic tree diagrams or hierarchical clustering. Figure 2.1.12 presents an example of the tree diagram [48].

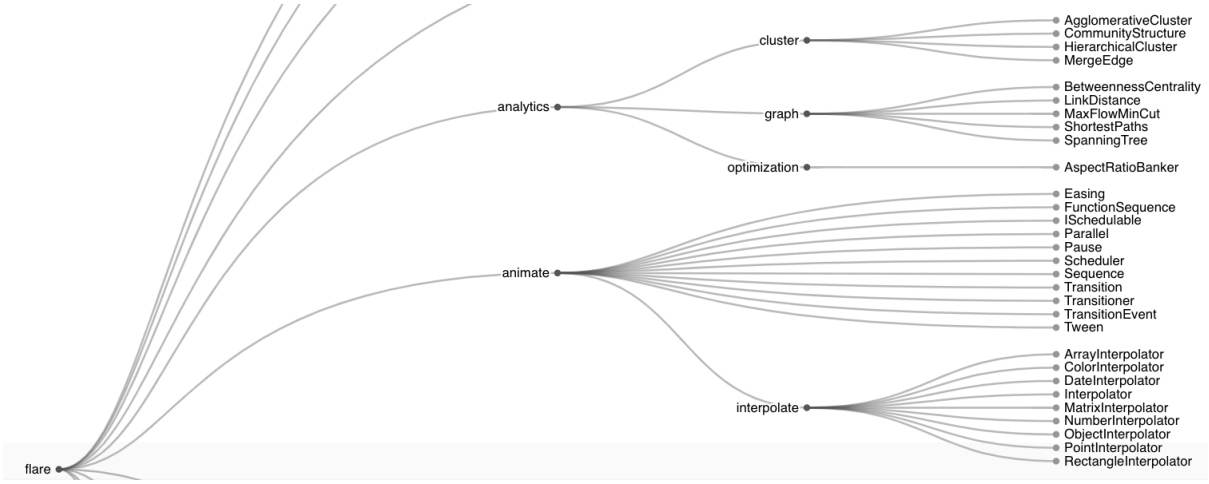


Figure 2.1.12 An example of using D3 cluster to create the tree diagram

A d3.cluster function sets out the particular root hierarchy, and the following properties on the root and its descendants are assigned,

- node.x - the node coordinate in x axis
- node.y - the node coordinate in y axis

Treemap

A treemap recursively segments an area into rectangles, as per the relevant value of each node. It was first developed by Ben Shneiderman in 1991 [49]. The establishment of a D3 treemap aids an extensible tiling method. The standard approach is to attempt to produce rectangles with a golden aspect ratio. The reason for this is that it enhances

readability and size estimation in comparison to slice-and-dice, which essentially just switches between horizontal and vertical segmentation based on depth. Figure 2.1.13 presents an example of the treemap diagram [50].

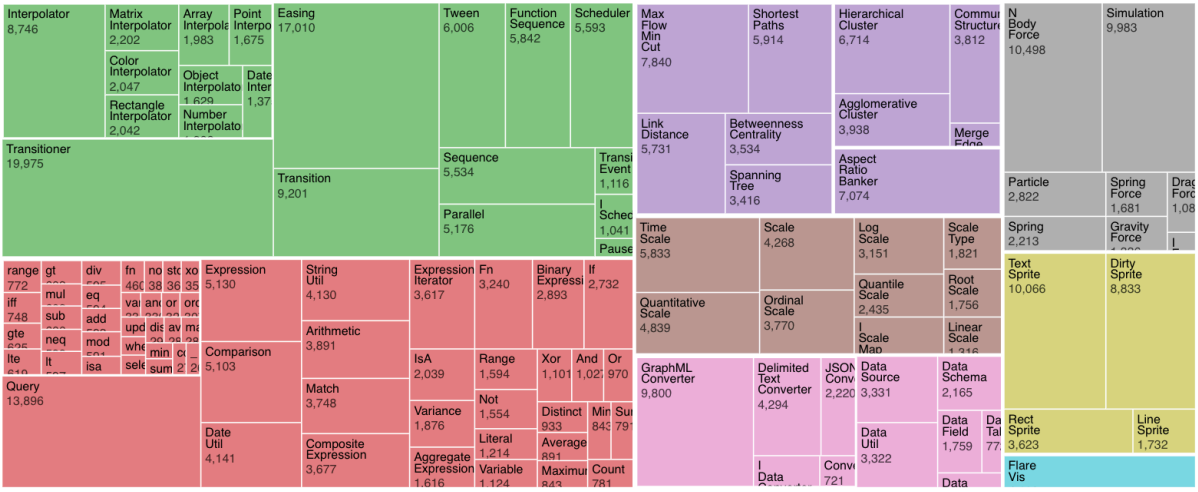


Figure 2.1.13 An example of using d3.treemap to create the treemap diagram

A d3.treemap sets out the particular root hierarchy, and the following properties on the root and its descendants are assigned,

- node.x0 - the node's left edge of the rectangle
- node.y0 - the node's top edge of the rectangle
- node.x1 - the node's right edge of the rectangle
- node.y1 - the node's bottom edge of the rectangle

Pack

Enclosure diagrams represents hierarchy by using containment or nesting. The leaf circles' size encodes a measurable facet of the data. The approximate cumulative size of each subtree is displayed by the enclosing circles; however, it is important to note that there is some wasted space, which causes a degree of misrepresentation. Hence, an accurate comparison can only be made with the leaf nodes. While circle packing is less

efficient then a treemap in terms of use of space, the wasted space actually better shows the hierarchical structure. Figure 2.1.14 presents an example of an enclosure diagram [51].

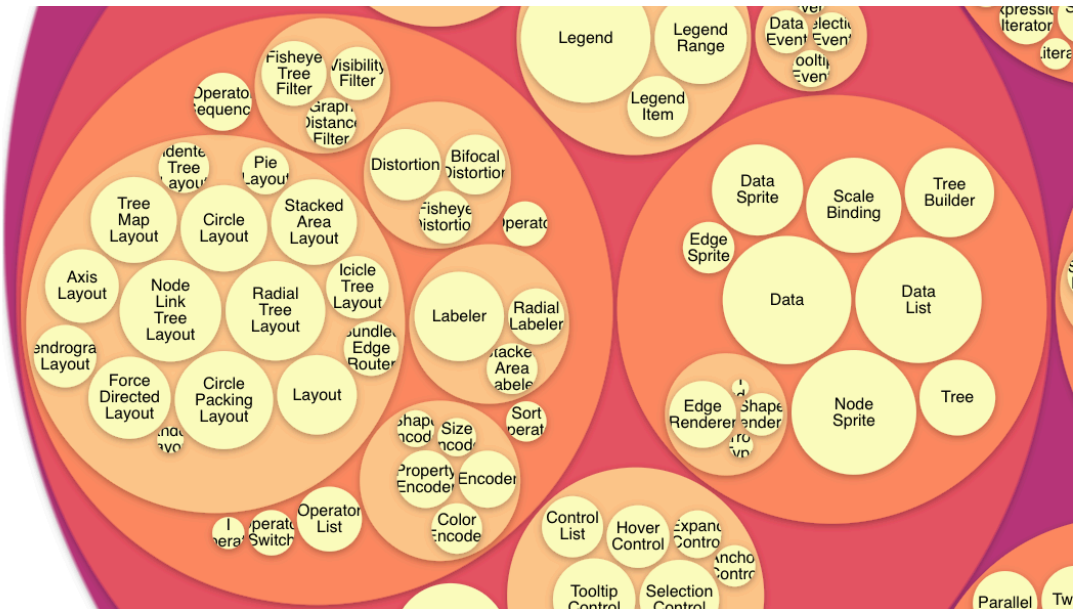


Figure 2.1.14 An example of using d3.pack to create an enclosure diagram

A d3.pack sets out the particular root hierarchy, and the following properties on the root and its descendants are assigned,

- node.x - the coordinate of the circle's centre in x axis
- node.y - the coordinate of the circle's centre in y axis
- node.r - the value of the circle's radius

D3 Geographies

Sometimes, point transformations are utilised to implement map projections. For example, the spherical Mercator,

```
function mercator(x, y) {
  return [x, Math.log(Math.tan(Math.PI / 4 + y / 2))];
}
```

If the geometry in question is comprised of continuous, infinite point sets, this is a sensible mathematical strategy. However, computers do not have infinite memory, therefore it is more appropriate to employ discrete geometry such as polygons and polylines.

Discrete geometry increases the difficulty of projecting from the sphere to the plane. Spherical polygon edges are geodesics, which means that they are segments of great circles rather than straight lines. With the exception of gnomonic map projections, when they are projected to the plane, geodesics are curves in all map projections. Therefore, in order to ensure precise projection, interpolation along each arc is necessary. D3 employs adaptive sampling that is based on a prevalent line simplification method so as to strike an effective balance between precision and accuracy [52].

Topological disparities between the sphere and the plane must be taken into account when projecting polygons and polylines. Cutting geometry that crosses the anti-meridian is necessary in some projections, and clipping geometry to a great circle is required in others.

Furthermore, in order to ascertain their internal side, spherical polygons require a winding order convention. When the polygon is smaller than a hemisphere, the exterior ring must be clockwise. Conversely, it must be anticlockwise when it is larger than a hemisphere. Interior rings that signify holes must employ a winding order that is opposite to that which is used in the exterior ring. TopoJSON [53] and ESRI [54]

shapefiles also utilise this winding order convention; in contrast, GeoJSON's RFC 7946 [55] does the opposite.

D3 provides significant flexibility, in that the user can select the correct projection and aspect based on the data at hand. Furthermore, D3 supports a broad range of map projections, both conventional and uncommon. In JavaScript, D3 utilises GeoJSON to embody geographic characteristics.

Geo Paths

There are comparisons between the shape generators in d3-shape and the geographic path generator, d3.geoPath. In a specific GeoJSON geometry or feature object, it either produces an SVG path data string or renders the path to a Canvas. It is advisable to use Canvas for interactive or dynamic projections, as it will enhance the performance. Paths are most appropriate for use with projections or transforms, or in circumstances when the user is rendering planar geometry directly to SVG or Canvas.

A new geographic path generator with default settings is created by the d3.geoPath. Subsequently, it renders the specific object, which could be any GeoJSON facet or geometry object, as follows,

- Point - a single position
- MultiPoint - an array of positions
- LineString - an array of positions creating a continuous line
- MultiLineString - an array of arrays of positions creating multiple lines
- Polygon - an array of arrays of positions creating a polygon (and holes)
- MultiPolygon - a multidimensional array of positions creating multiple polygons
- GeometryCollection – a collection of geometry objects
- Feature - a feature containing one of the above geometry objects
- FeatureCollection – a collection of feature objects

The sphere type is also supported. As a sphere does not have coordinates, this is advantageous for rendering the globe outline. Any further arguments are passed to the pointRadius accessor.

Raw Projections

Point transformation functions that are employed to carry out custom projections are referred to as raw projections. Conventionally, they are passed to d3.geoProjection or d3.geoProjectionMutator. At this point, they are exposed to enable the origination of associated projections. Raw projections take spherical coordinates in radians and return a point [x, y]. This is usually in the unit square that is surrounding the origin.

A new projection is created from the specified raw projection project by d3.geoProjection. This is done by taking the longitude and latitude of a specific radian point, which are frequently referred to as λ (lambda) and ϕ (phi), and this returns a two-element array [x, y], which signifies its unit projection. It is not necessary for the project function to scale or translate the point, because projection.scale, projection.translate, and projection.centre apply them automatically. Similarly, the project function is not required to conduct any spherical rotation, as prior to the projection, projection.rotate is applied.

Spherical Shapes

The process to produce a great arc, which is a segment of a great circle, is simple and straightforward. The GeoJSON LineString geometry object can be simply passed to a d3.geoPath; there is no requirement for an arc shape generator, because D3 projections employ great arc interpolation for intermediate points. Figure 2.1.15 presents a D3 generated example of the graticules [52].

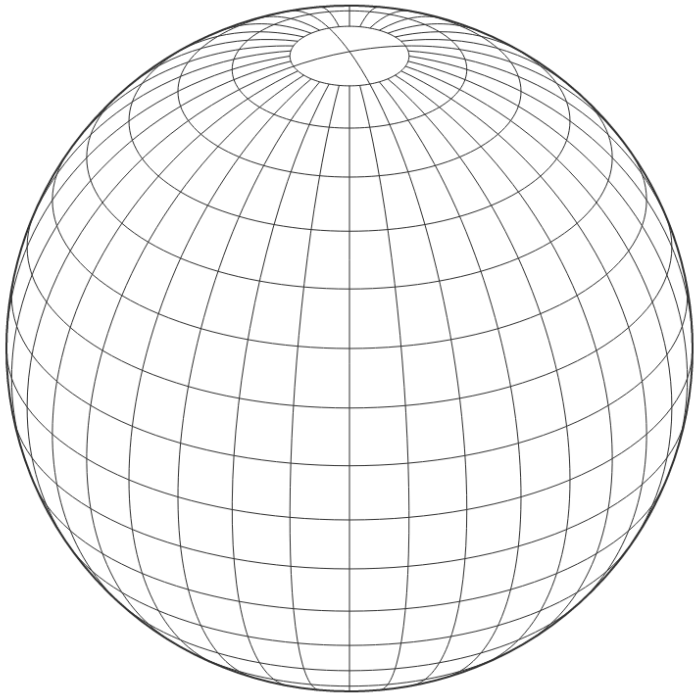


Figure 2.1.15 An example of using d3.geoGraticule to generate the graticules

d3.geoGraticule builds a geometry generator that produces graticules. Graticules are a homogenous grid of meridians and parallels that demonstrate projection distortion. The default graticule has meridians and parallels every 10° between ±80° latitude. However, there are meridians every 90° for the polar regions.

Steams

D3 employs a sequence of function calls to transform geometry, which minimises overhead in comparison to using materialising intermediate representations. It is essential that streams establish a variety of methods to receive input geometry. By nature, streams are stateful. This means that the meaning of a point is based on whether it is positioned inside a line. Similarly, a line and a ring are differentiated by a polygon. At present, these method calls are synchronous, regardless of being referred to as a 'stream'.

d3.geoStream streams the selected GeoJSON object to the particular projection stream. Although the features and geometry objects are both supported as input, the stream interface details the geometry only. Therefore, any additional feature traits are undetectable to streams.

Transforms

Transforms are defined as a generalization of projections. They carry out projection.stream and can be passed to path.projection. It is important to note though, that they only execute a subsection of the other projection methods. Furthermore, they represent random geometric transformations rather than projections from spherical to planar coordinates.

d3.geoTransform delineates a random transform utilising the approaches defined on the selected methods object. Pass-through methods that transmit inputs to the output stream are applied to any undefined methods. For instance, to reflect the y-dimension by using the following transform,

```
var reflectY = d3.geoTransform({
  point: function(x, y) {
    this.stream.point(x, -y);
  }
});
```

2.1.5 Summary

In the digital age of today, data visualisation has become an essential discipline as data continues to grow as a new resource in modern society that has very unique values. Data visualisation is classified as exploratory and explanatory depending on its purpose. The data visualisations methodology is also classified into seven methods, which are comparing categories, assessing hierarchies, presenting changes overtime, plotting connections and relationships, and mapping geo-spatial data. In terms of presenting data visualisations, compared to the traditional static visualisations such as print, web-based interactive visualisations are becoming increasingly popular and essential in today's daily lives.

The modern web technologies can help in delivering highly interactive data visualisations in web-based environments, ranging from an immersive data visualisation observatory to a pocket-sized smartphone. Such flexibility is enabled by the very fundamental elements of HTML, the cascading mechanism of CSS, and the highly adaptive scripting language of JavaScript, wherein JavaScript is featured according to its capability of manipulating dynamic object properties. Moreover, the wide support of DOM that JavaScript provides in various popular browsers also play a vital role in enabling various events and functions because modern browsers are commonly developed with the intention of them being compatible with various devices.

D3.js is a powerful data visualisation library that allows the realisation and creation of novelty data visualisations from imagination. Its selections and data functions aid in effective data-driven transformation of the DOM, while allowing data to be easily manipulated using selected elements. D3 scales and axes can simplify the fundamental and common tasks concerning visualisation for developers. The transition and timer modules also help to animate changes to the DOM while keeping the concurrent animations synchronised and consistent. Further, D3 shapes and colours aid the developers to conveniently develop the desired geometry with colours, along with the

D3 hierarchies and geographies that are the more advanced modules to create relatively complicated and structured diagrams and maps.

2.2 Distributed Systems

2.2.1 Introduction

Although there are several definitions of distributed systems in the literature, not one of them is adequate or agrees with the other definitions. In terms of this thesis, a general characterization is sufficient. A distributed system can be considered as a collection of independent computers which seems to be a single coherent system to its users [56].

Distributed systems tend to be complicated software whose components are, as per the definition, distributed throughout numerous machines. It is important to thoroughly organise these systems so that their complexity can be mastered. A distributed system's organisation can be perceived in diverse ways. A clear method is distinguishing between the software component collections' logical organization and the actual physical realisation.

Distributed systems' organisation largely concerns the software components that form the system. Such software architectures indicate the organisation of the diverse software components and the way in which they must interact. It is important that the software components are incorporated and established on real machines to ensure actual realisation of a distributed system. For this, various methods can be implemented. Moreover, a software architecture's final instantiation is also called a system architecture.

The following sections will examine traditional centralised architectures wherein one server executes the majority of the software components, and hence also functionality, and where remote clients are able to gain access to that server through simple communication means. Further, decentralised architectures where machines are largely equal, and hybrid organizations will also be examined.

It is vital to use such a layer in terms of architectural decision, and it primarily intends to ensure distribution transparency. On the other hand, trade-offs must be made for attaining transparency, resulting in diverse methods of ensuring that middleware is adaptive. A latter part of this section will explore some more commonly applied ones that impact the middleware organisation.

2.2.2 Architecture Designs

2.2.2.1 Centralised Design

Although there is no consensus regarding the several problems of distributed systems, multiple practitioners agree that it can be helpful to think that *clients* request services from *servers* as it can address the complicated nature of distributed systems.

The basic client-server model depicts the processes in a distributed system being classified into two groups that may overlap. A server refers to a process that uses a particular service such as a database service or a file system service. A client refers to a process requesting from a server a service for which a request is sent, and the server's reply is awaited. Figure 2.2.1 illustrates this client–server interaction which is also called request–reply behaviour [56].

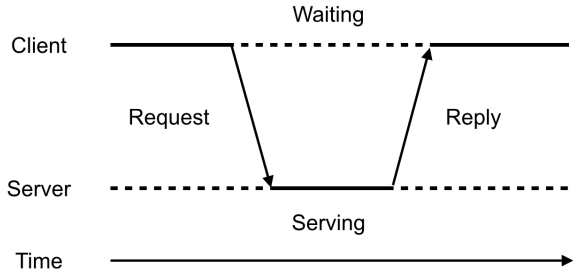


Figure 2.2.1 An illustration of the common interaction between a client and a server

A simple connectionless protocol can be used for enabling communication between client and server, similar to several local-area networks, the underlying network is quite reliable. Thus, when a service is being requested by a client, the process involves a message being packages for the server and the intended service being determined as well as the required input data. Following this, the message is sent to the server which always awaits an incoming request, then processes it, and creates a reply message with the results which it sends to the client.

The benefit of using a connectionless protocol is that it is efficient. The request/reply protocol that has been sketched works well until the messages are not lost or corrupted. However, it is important to ensure that the protocol can resist occasional transmission failures. The possible solution here when there is no reply is to allow the client to resend the request. The fact that the client is unable to identify if the reply's transmission failed or if original request message was lost is a problem. Resending a request when the reply is lost can lead to the operation being performed twice. In case the operation was similar to 'transfer some cash from my bank account', reporting an error will be a better response. In case, however, the operation was 'tell me how much money I have left', resending the request will be more suitable. An operation is considered idempotent when it is can be repeated without adverse effects multiple times. As certain requests are idempotent while others are not, it is evident that no particular solution can be implemented to address the lost messages.

The alternative solution is that several client–server systems implement a reliable connection-oriented protocol. Despite this solution not being completely suitable in a local-area network because of relatively poor performance, it works well in wide-area systems with inherently unreliable communication. Almost all Internet application protocols, for example, are founded on dependable TCP/IP connections. Here, every time a service is requested by a client, a connection is first established with the server before the request is sent. Typically, the server makes use of that same connection for sending the reply, and then the connection is eliminated. Establishing and eliminating a connection is, however, expensive, particularly in cases where the request as well as reply messages are small.

In recent years, there have been numerous debates regarding the client–server model. A major problem concerns how a clear distinction can be made between client and server. It has often been noted that no clear distinction is possible. A server regarding a distributed database, for example, can continuously function as a client as it forwards

requests to various file servers that implement the database tables. Here, the database server itself only processes the queries.

However, as several client–server applications focus on enabling user access to databases, it has commonly been recommended that a distinction be made between the user-interface level, the processing level, and the data level, particularly considering the layered architectural style previously explored.

The user-interface level includes everything, such as display management, required for interfacing directly with the user. Usually, the processing level includes the applications. Moreover, the data level is responsible for managing the data being acted on. The user –interface level is generally used by clients typically implement the user-interface level. At this level, programs which help end users' interaction with applications are included. The level of sophistication between user-interface programs is considerable.

A character-based screen is the simplest user-interface program. This type of interface is often used in mainframe environments. When the mainframe is in controls of all interaction including monitor and keyboard, a client–server environment is not very likely. On the other hand, there are several cases where the user's terminal performs certain local processing including typed keystrokes being echoed or providing support to form-like interfaces wherein an entry which is complete must be edited prior to it being sent to the main computer.

Today, advanced interfaces are included even in mainframe environments. The client machine usually provides a graphical display that uses pop-up or pull-down menus and where several screen controls are used through a mouse rather than keyboard. Examples of these interfaces are the X-Windows interfaces that several UNIX environments use, and earlier interfaces which were developed for Apple Macintoshes and MS-DOS PCs.

As modern user interfaces enable applications in which a single graphical window is shared that can be used for data exchange through user actions, it can provide more functionality. For example, it deletes a file by moving the icon which represents the file to another icon which represents a trash can. Similarly, in several word processors, users can move text within a document by using the mouse alone.

It is possible to construct several client–server applications using approximately three distinct pieces which are a part which takes care of user interaction, a part which operates on file system or a database, and a part which tends to include an application's core functionality. This latter part is placed logically at the processing level. Compared to databases and user interfaces, few aspects are common to the processing level. Hence, several examples will be provided so that this level is clearer.

Consider for example, an Internet search engine. Apart from the animated banners and images, a search engine has a simple user interface in which a string of keywords are typed in by the user and a list of Web pages' titles is then provided to them. The back end is developed through a substantial database of prefetched and indexed Web pages. Central to the search engine is a program which can transform the user's keywords into database queries. Following this, the results are ranked into a list and this list is changed into HTML pages. In the client–server model, as shown in Figure 2.2.2, this information retrieval part tends to be included at the processing level [56].

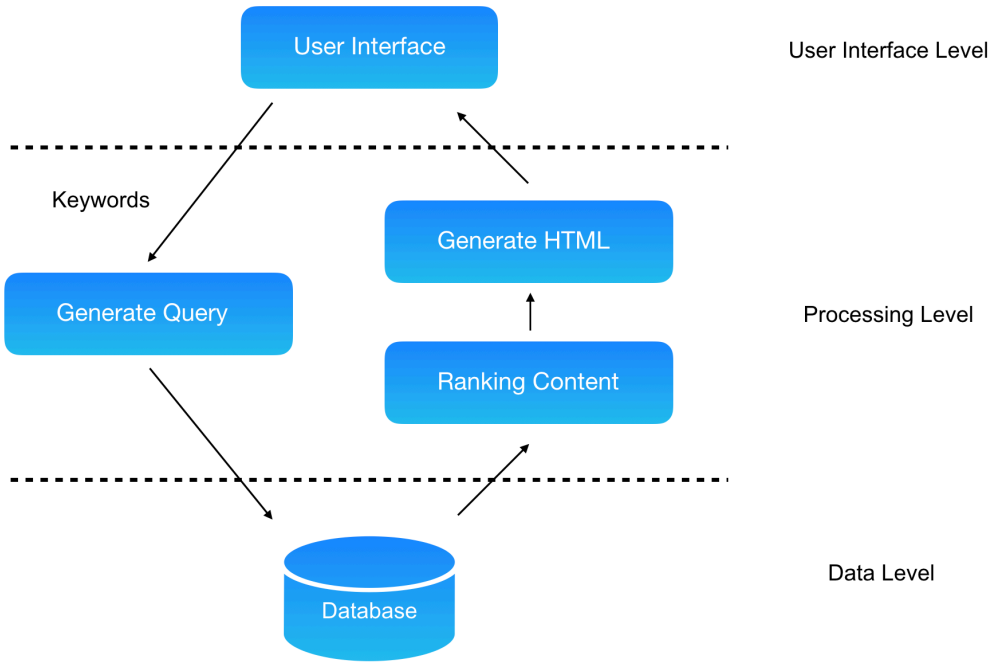


Figure 2.2.2 The three levels of a simplified Internet search engine

In the client–server, the data level includes the programs responsible for maintaining the actual application-operated data. At this level, data tends to be persistent so that data will continue to be stored to be used again despite there being no application running. The data level in its simplest form includes a file system, though using a complete database is more common. The data level in the client–server model tends to be implemented at the server side.

Apart from storing data, the data level also ensures that data is consistent in various applications. In cases where databases are used, sustaining consistency includes storing metadata such as entry constraints, application-specific metadata, and table descriptions. For example, regarding a bank, it may be important to develop a notification for a customer's credit card debt reaching a specific value. A database trigger can help in maintaining such information as it can activate a handler at the suitable moment.

The data level is structured similar to a relational database in the majority of business-oriented environments. Here, data independence is important. The organisation of the data does not depend on the applications so that organisational changes do not impact applications and the applications do not impact the data organization. It can be helpful to use relational databases regarding the client–server model for differentiating between the data level and processing level as they are both independent.

On the other hand, relational databases cannot always be considered as the ideal choice. A significant feature of several applications is their operation on complicated data types which can be modelled better regarding objects rather than relations. Some examples of this type are simple polygons and circles and aircraft designs representations such as computer-aided design (CAD) systems [57].

When data operations can be expressed better using object manipulations, implementing the data level through an object-relational or object-oriented database is a more appropriate idea. It is important to note that the object-relational database type has been popular because such databases expand the relational data model which is widely dispersed and provide the object-orientation advantages.

The differentiation between the three logical levels as previously discussed indicates several possibilities concerning physical distribution of a client–server application through multiple machines. Having two types of machines is the simplest organization structure.

- The first is a client machine which includes only those programs that implement part of the user-interface level
- The second is a server machine which includes the programs that implement the data level and processing.

The server is responsible for everything in such an organization and the client is only a terminal, with perhaps a suitable graphical interface. Several other possibilities are available, and the ones that are more common are explored in this section.

One of the approaches to organise the servers and clients is distributing the programs in the previous section's application layers throughout various machines, as illustrated in Figure 2.2.3 [56]. The first step is to differentiate between the two types of machines which are client machines and server machines, also called a physically two-tiered architecture.

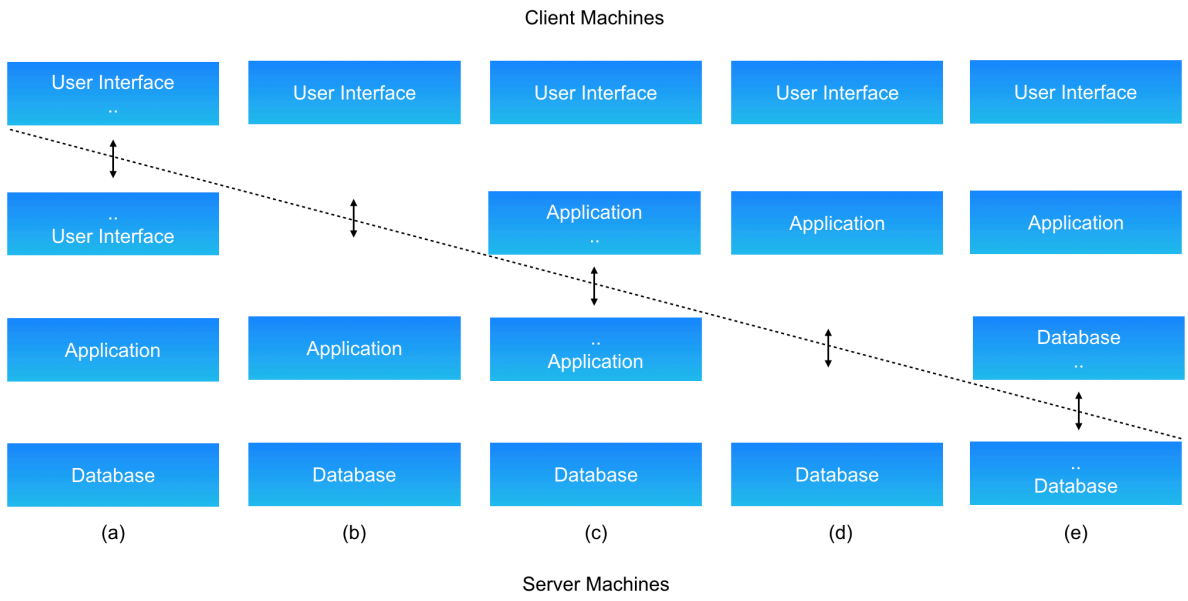


Figure 2.2.3 The alternative client-server organisations

A possible organization is, as shown in Figure 2.2.3 (a), the client machine having only the terminal-dependent part of the user interface, while the applications have remote control over their data presentation. Another option is the client side having the complete user-interface software, as indicated in Figure 2.2.3 (b). If this is the case, the application is divided into a graphical front end that communicates with the other application part at the server using an application-specific protocol. The front end or the

client software in this model does not conduct any processing apart from what is required to present the application's interface.

Moreover, it is also possible to move some aspects of the application, as illustrated in Figure 2.2.3 (c), to the front end. An example of this is when the application uses a form which must be completely filled before being processed. Then, the front end can verify the form's consistency and accuracy as well as interact with the user if required. An example of Figure 2.2.3 (c) organisation structure is a word processor wherein the fundamental editing functions are conducted on the client side operating on locally cached or in-memory data, whereas the advanced support tools including checking the grammar and spelling is executed on the server side.

The organizations presented in Figure 2.2.3 (d) and Figure 2.2.3 (e) are quite popular in several client–server environments. Such organizations are used in case the client machine is a workstation or a PC, connected to a distributed file system or database through a network. While the majority of the application tends to run on the client machine, operations that run on files or on database entries go to the server. Several banking applications, for example, function on a machine of the end-user where transactions are prepared. After this is done, the application makes contact with the database on the server of the bank for uploading the transactions to be processed further. Figure 2.2.3 (e) shows a scenario in which the client's local disk includes part of the data. For example, when a client is browsing the Web, they may gradually develop a significant cache on the local disk regarding the Web pages that were most recently examined.

In the past few years, it has been commonly seen that the configurations presented in Figure 2.2.3 (d) and Figure 2.2.3 (e) have not been implemented when client software is placed at end-user machines. Here, the majority of the processing as well as data storage is taken care of at the server side because even if client machines perform many functions, they can be difficult to manage. The client machine having more functionality

results in the client-side software becoming more prone to errors as well as more reliant on the underlying platform of the client which includes operating system and resources. Considering a system's management, it is not the best to have so-called fat clients and better to have thin clients, as shown in the organisations in Figure 2.2.3 (a) to (c), which may lead to less-sophisticated user interfaces and client-perceived performance.

It should be noted that this trend is not suggesting that distributed systems are not needed. In fact, it shows that server-side solutions are becoming more distributed because multiple servers that function on various machines are replacing a single server. Particularly, when differentiation between only client machine and server machine, it is often overseen that a server can have to function as a client, as illustrated in Figure 2.2.4, resulting in a physically three-tiered architecture. Programs in this architecture which are part of the processing level are placed on a separate server, though they may be partly distributed through the client as well as server machines. Transaction processing is an example of the use of a three-tiered architecture.

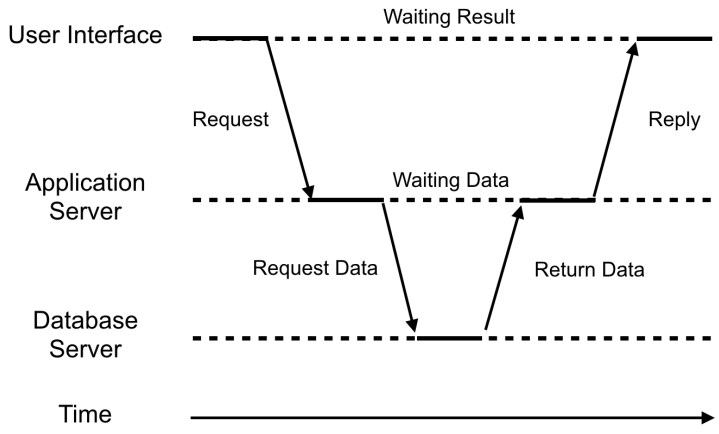


Figure 2.2.4 An example of a server acting as a client

Moreover, the organisation of Web sites is another example of a three-tiered architecture. Here, a Web server functions a place of entry to a site which passes requests to an application server so that the actual processing can occur. Then, this

application server and a database server interact. An application server, for example, can run the code for examining the available inventory concerning some goods which an electronic bookstore offers. For this, it may have to interact with a database that includes the raw inventory data.

2.2.2.2 Decentralised Design

When applications are classified into processing components, data level, and user-interface, it leads to multitiered client-server architectures. The various tiers are equivalent to the applications' logical organisation. Regarding several business environments, distributed processing is similar to a client-server application being organised as a multitiered architecture. Such a distribution is called vertical distribution. Further, a characteristic feature concerning vertical distribution is that it can be attained by positioning *logically* different components on various machines. This term concerns the concept of vertical fragmentation the way it is implemented in distributed relational databases, in which tables are divided according to columns and then distributed throughout diverse machines.

In addition, in terms of system management, it can be helpful to have a vertical distribution, and functions are divided logically as well as physically throughout several machines, with every machine being customised as per a particular group of functions. On the other hand, there are numerous other ways apart from vertical distribution to organise client-server applications. Regarding modem architectures, distribution of the clients as well as the servers is often what matters, which is called horizontal distribution. A client or sever in such a distribution can be divided physically into logically equivalent parts, with every part functioning as its own of part the complete data set, and hence, the load is balanced. This section will examine a class of modern system architectures called peer-to-peer systems which supports horizontal distribution.

Considering a high-level perspective, the peer-to-peer system processes are all equal; that is, the functions necessary to be executed are represented by processes that form the distributed system. Because of this, the interaction between processes was symmetric to a large extent with every process functioning like a client as well as a server at the same time. This is also called acting as a servant.

Peer-to-peer architectures in terms of this symmetric behaviour progress concerning how the processes can be organised in an overlay network, which is a network where the processed form the nodes and the links indicate the possible communication channels that are typically developed by TCP connections. Overall, a process is unable to directly communicate with another arbitrary process while being required to use the available communication channels to send messages. There are two types of overlay networks, structured and unstructured. Lua et a1. [58] extensively surveyed these two overlay networks using several examples. Moreover, Aberer et al. [59] provides a reference architecture which enables the various peer-to-peer systems to be compared more formally.

Structured Peer-to-Peer Architectures

A deterministic procedure is used to develop the overlay network in a structured peer-to-peer architecture. Organising the processes using a distributed hash table (DHT) is by far the procedure that has been used the most. Furthermore, a random key is assigned to data items in a DHT -based system from a large identifier space, such as a 160-bit or 128-bit identifier. Similarly, a random number is also allocated to nodes in the system from the same identifier space. Thus, every DHT-based system's crux is implementing an effective as well as deterministic scheme which maps a data item key uniquely to a node identifier as per certain distance metric [60]. In addition, when a data item is being looked up, the network address of the node in charge of that data item is returned. This is efficiently conducted by *routing* a data item request to the responsible node.

In the Chord system [60], for example, the organisation of nodes in a ring is executed logically so that the mapping of a data item that has key k is done to the node that has the smallest identifier $id \sim k$. This node is called key k's *successor* and is referred to as *succ(k)* in Figure 2.2.5. For looking up the data item, an application that is being

executed on an arbitrary node will call the function *lookup*(k) that will return the *succ(k)*'s network address. Then, the application is able to contact the node and attain a copy of the data item.

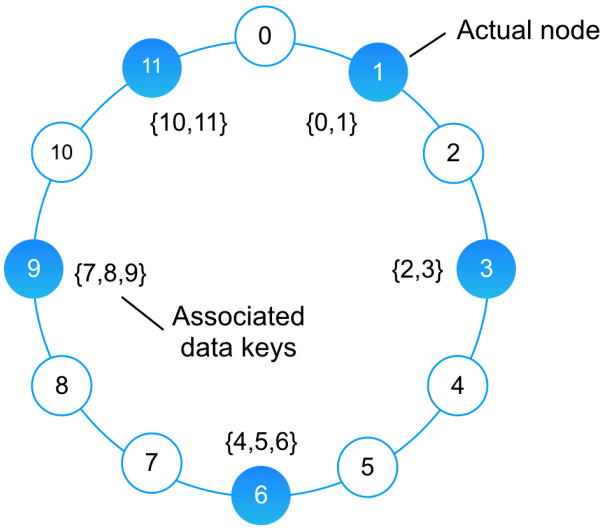


Figure 2.2.5 The mapping of data items on nodes in Chord

Focusing on the way in which nodes organise themselves to form an overlay network, or a membership management, it is necessary to acknowledge that looking up a key is not in accordance with the nodes' logical organisation in the ring, as shown in Figure 2.2.5. Nevertheless, every node retains shortcuts that allow access other nodes so that lookups can usually be conducted in *O(log (N))* number of steps, with *N* indicating the number of nodes involved in the overlay.

If we consider Chord again, a node that wants to join the system begins with developing a random identifier *id*. It should be noted that in case of the identifier space being sufficiently large, if there is a good quality of random number generator, then there is close to zero probability of developing an identifier which is already assigned to an actual node. The node can then conduct a lookup on *id* because of the network address of *succ(id)* will be returned. Here, the joining node can contact *succ(id)* as well as its predecessor while placing itself in the ring. Such a scheme undeniably needs every node

to store information about its predecessor. Further, insertion indicates that every data item with key that is related to node *id* is transferred from *succ(id)*.

Unstructured Peer-to-Peer Architectures

It is remarkable that peer-to-peer systems that are unstructured tend to depend on randomised algorithms to develop an overlay network. The central concept is that although every node retains a list of neighbours, the list is generated in a random manner. Similarly, it is assumed that data items are randomly positioned on nodes. This results in nodes flooding the network with a search query when it has to locate a particular data item [61].

Several unstructured peer-to-peer systems aim to develop an overlay network which is similar to a random graph. The fundamental model includes every node retaining a list of c neighbours, in which every neighbour depicts a *live* node that is selected randomly from the current nodes set. This list of neighbours is also called a partial view which can be developed in various ways. A framework was created by Jelasity et al. [62][63] which includes various algorithms concerning overlay construction that enabled comparisons as well as assessments. This framework assumed that entries are consistently exchanged by nodes from their partial view. Another node is also identified by every entry in the network, with each entry having a related age suggesting how old the node reference is.

Figure 2.2.6 [56] shows that the active thread initiates the communication with another node by choosing that node from its existing partial view. If the entries have to be *pushed* toward the selected peer, it develops a buffer that includes *c/2* + I entries which also includes an entry that identifies itself. The existing partial view is from where other entries are selected.

Generating a new partial view is critically important. The view, in terms of initiation and as of the contacted peer, includes precisely *c* entries that will partially be derived from received buffer. The new view can be essentially constructed in two ways. The first is where the two nodes can choose to get rid of the entries which they had sent to one another, which indicates that some of their original views will be *swapped*. The second is where *old* entries are discarded as much as possible. The two approaches seem to generally be complementary. It has been noted that this framework includes several membership management protocols concerning unstructured overlays. Several interesting observations can be made here.

```
Actions by active thread:
// select a peer  P from the current partial view;
if PUSH_MODE {
  mybuffer = [(address, 0)];
  permute partial view;
  move H oldest entries to the end;
  append first c/2 entries to my buffer;
  send mybuffer to P;
} else {
  send trigger to P;
}
if PULL_MODE {
  receive P's buffer;
}
// construct a new partial view from the current one and P's buffer;
increment the age of every entry i the new partial view;
```

(a)

```
Actions by passive thread:
// receive buffer from any process Q;
if PULL_MODE {
  mybuffer = [(address, 0)];
  permute partial view;
  move H oldest entries to the end;
  append first c/2 entries to my buffer;
  send mybuffer to P;
}
// construct a new partial view from the current one and P's buffer;
increment the age of every entry in the new partial view;
```

(b)

Figure 2.2.6 Actions of the active thread (a) Actions of the passive thread (b)

First, assuming that a node gets in contact with another arbitrary node when intending to join which may be from numerous well-known access points, this access point can be regarded as a regular part of the overlay, apart from the fact that it can be regarded as being highly available. If this is the case, protocols using only *push* or only *pull* mode can result in disconnected overlays quite easily. That is, groups of nodes will be isolated and will be unable to contact each node in the network. This is an obviously undesirable feature and thus nodes must be allowed to *exchange* entries.

Second, a fairly straightforward operation is required to leave the network if partial views are regularly exchanged by the nodes. Here, it is possible for a node to leave and not inform other nodes. Because of this, a node P choosing an apparent neighbour, let us assume node Q, and determining that Q is not responding anymore, it will eliminate the entry and choose another peer from its partial view. Thus, when a new partial view is being developed, a node functions in tandem with the policy for removing as many old entries as it can, and the nodes that leave are quickly forgotten. That is, entries concerning departed nodes are quickly and automatically removed from partial views.

On the other hand, following this strategy also has its consequences. Assume a node *P* having a set of nodes' partial view having an entry referring to *P.* This is called the indegree of a node. Id node P's indegree is higher, it will lead to higher probability of another node contacting *P.* That is, *P* is at risk of becoming a popular node that can easily create an imbalanced position concerning workload. When old entries are systematically discarded, nodes are promoted into ones with high indegree. Other trade-offs also exist which are explored by Jelasity et al. [63].

Super-peers

It should be noted that, in peer-to-peer systems that are unstructured, it can be difficult to locate relevant data items because of growing network. This scalability problem is because of there being no deterministic method to route a lookup request to a particular data item, resulting in the node only being able to flood the request. Though it is possible to dam such flooding in numerous ways, several peer-to-peer systems have suggested using special nodes as an alternative as they maintain an index of data items.

Leaving the peer-to-peer systems' symmetric nature is also practical in other situations. Examine a collaboration of nodes providing one another with resources. Considering a collaborative content delivery network (CDN), for example, nodes can provide storage to host Web pages' copies that would enable Web clients to gain access to nearby pages so that they can swiftly access them. Here, it is possible that a node *P* has to search for resources in a particular part of the network. Hence, it will be helpful to use a broker which gathers resource usage for numerous nodes that are close to each other as it enables swift selection of a node through adequate resources.

Typically, nodes that are similar to ones that function as a broker or maintain an index are called super-peers. Super-peers, as indicated by their name, tend to be organised in

a peer-to-peer network which results in a hierarchical organization, as suggested by Yang and Garcia-Molina [64]. Figure 2.2.7 illustrates one such example of an organization where each regular peer is connected to a super-peer as a client. Communication taking place from and to a regular peer is conducted using that peer's associated super-peer.

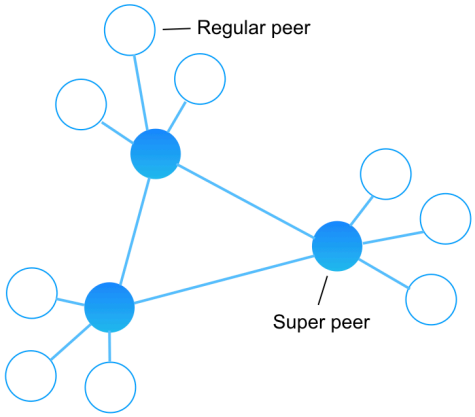


Figure 2.2.7 An organisation of nodes in a super-peer network

The client-super-peer relation is established in several cases. That is, every time a regular peer is included in the network, that peer gets connected to a super-peer and continues to be connected until it leaves the network. Super-peers are thus expected to be long-lived processes and having high availability. It is possible to compensate for a super-peer's potential unstable behaviour by implementing backup schemes, including each super-peer being paired with another super-peer and needing clients to attach to both.

A super-peer with which an established association is made is not always the optimum solution in all instances. Take, for example, the file-sharing networks. In this case, getting attached to a super-peer maintaining a file index in which the client is interested may be a better solution for a client. This will provide better chances compared to the client seeking a particular file as its super-peer will be able to find it quicker. Garbacki

et al. [65] proposed a scheme that is relatively simple wherein the relationship between the client and super-peer can change because clients find better super-peers with which to get attached. That is a super-peer that is providing a lookup operation's results is prioritised over other super-peers.

2.2.2.3 Hybrid Design

The focus so far has been on various peer-to-peer architectures and client-server architectures. In several distributed systems, architectural features are merged, as seen in super-peer networks. This section will examine certain particular classes of distributed systems where client-server solutions and decentralised architectures are combined.

Edge-server systems develop a crucial class of distributed systems organised as per a hybrid architecture. Such systems are positioned on the Internet such that servers are placed 'at the edge' of a given network. Such an edge is developed through the boundary that exists between actual Internet and enterprise networks, such as that provided by an Internet Service Provider (ISP). Similarly, in cases of end users who are at home using their ISPs to connect to the Internet, that ISP can be regarded as being positioned at the edge of the Internet. From this, a general organization can be formed which is shown in Figure 2.2.8.

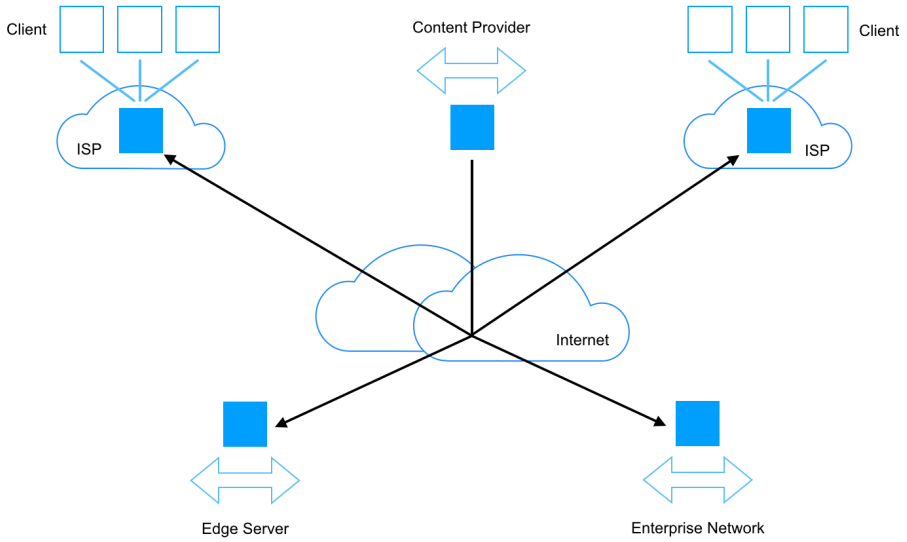


Figure 2.2.8 The Internet that consists of a collection of edge servers

End users, or typical clients, use an edge server to connect to the Internet. This edge server primarily aims to provide content, feasibly after the content is filtered and the functions transcoded. It should be noted that it is possible to use a collection of edge servers for enhancing content as well as application distribution. The fundamental model involves an edge server for a particular organization functioning as an origin server that is the origin for all content.

Collaborative distribution systems are known to have hybrid structures deployed in them. The major problem for several such systems is to get started, and a traditional client-server scheme is often deployed for this. When a node connects to the system, it is able to utilise a fully decentralised scheme to collaborate.

To further establish this, take the BitTorrent file-sharing system into account [66]. BitTorrent can be referred to as a peer-to-peer file downloading system whose principal working is illustrated in Figure 2.2.9. In this system, when an end user seeks a file, chunks of the file are downloaded by the end user from other users until it is possible to assemble the downloaded chunks to create the complete file. Ensuring collaboration is a crucial design goal. Although a large number of participants in the majority of file-

sharing systems only downloaded files, they contribute almost nothing else [67][68][69]. Hence, it is possible to download a file only when the client who is downloading is offering content to someone else.

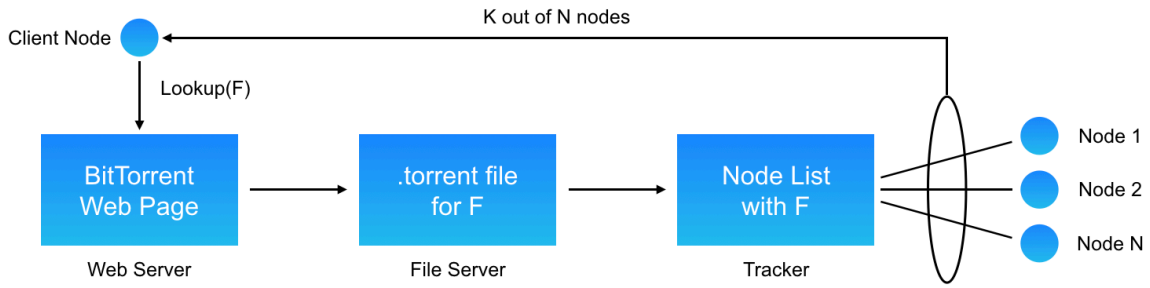


Figure 2.2.9 An illustration of the working mechanism of BitTorrent

It is crucial that a user accesses a global directory that forms a well-known Web site. This directory includes references to *.torrent* files which contain information required for downloading a particular file. That is, it indicates what is called a tracker that is a server which maintains a precise account of *active* nodes containing chunks of the requested file. A note that is presently downloading another file is called an active node. Hence, though there will undeniably be several trackers, typically only a single tracker for every file (or collection of files) exists.

After determining the nodes from which chunks are downloaded, the node download becomes active. It is then compelled to aid others by, for example, offering chunks of the file that is being downloaded which is not provided to others yet. Such enforcement results from a simple rule which states that in case of node P observing node Q downloading more what it is uploading, P can reduce the rate at which it sends data to Q. Such a scheme is successful only if P has to download something from Q. This is why nodes tend to be provided with references to various other nodes which gives them an advantage for trading data. Hence, BitTorrent merges centralised and decentralised solutions. It is observed that the system's bottleneck is developed by the trackers.

The Globule collaborative content distribution network [70] is another example. Globule has a compelling resemblance with the edge server architecture previously mentioned. Here, rather than edge servers, end users as well as organizations provide enhanced Web servers voluntarily which can collaborate in replicating Web pages. Every such server includes,

- A component capable of redirecting client requests to other servers,
- A component that can assess patterns,
- A component that can replicate Web pages.

The server that Alice provides is the Web server which typically deals with Alice's Web site traffic and is known as that site's origin server. This server works together with other servers, such as the one Bob provides, for hosting Bob's site pages. This, Globule can be considered as a decentralised distributed system. At first, requests concerning Alice's Web site get sent to her server, after which it is possible that they are redirected to another server. Distributed redirection may also be implemented.

However, a centralised component also exists in Globule as its broker which registers servers and makes others known such servers. An analogous communication exists between servers and the broker, as against what may be expected from a client-server system. Because of availability, it is possible to replicate the broker, and as observed eventually in this book, such replication is implemented commonly for ensuing valid client-server computing.

2.2.3 Architectural Styles

Considering the architecture in terms of a further abstraction from different architecture designs, an architectural style becomes important. This style is developed considering components in term of all the components connected, the data exchanged between components, and the way in which these elements are together configured into a system. A component refers to a modular unit that has well-defined needed and provided interfaces which can be replaced within its environment [71].

As examined below, a central problem concerning a distributed systems' component is that it is replaceable, once its interfaces are respected. Moreover, a concept that may be difficult to comprehend is a connector that is often referred to as a mechanism which mediates communication, cooperation, or coordination among components [72][73]. A connector, for example, can be developed by the facilities concerning (remote) procedure calls, streaming data, or message passing.

Considering connectors and components, different configurations can be attained that have been divided into architectural styles. Thus far, numerous styles have been determined. Of these, there are four that are crucial for distributed systems which are layered architectures, object-based architectures, data-centred architectures, and event-based architectures.

There is a simple concept concerning the fundamental idea for the layered style which is that the organisation of components is conducted in a layered manner so that a component at layer *L* is permitted to call components at the underlying layer but it does not work the other way around (Figure 2.2.10 (a)). Such a model has been implemented extensively in the networking community. It has also been noted that control tends to flow from one layer to another and requests flow downwards in the hierarchy while the results move in an upward direction.

Object-based architectures, shown in Figure 2.2.10 (b), follow a more flexible organisation. Here, every object is parallel to a component, with the components connected using a (remote) procedure known as mechanism. Hence, such software architecture is similar to the above-described client-server system architecture. Regarding large software systems, the layered and object-based architectures remain the most crucial [74].

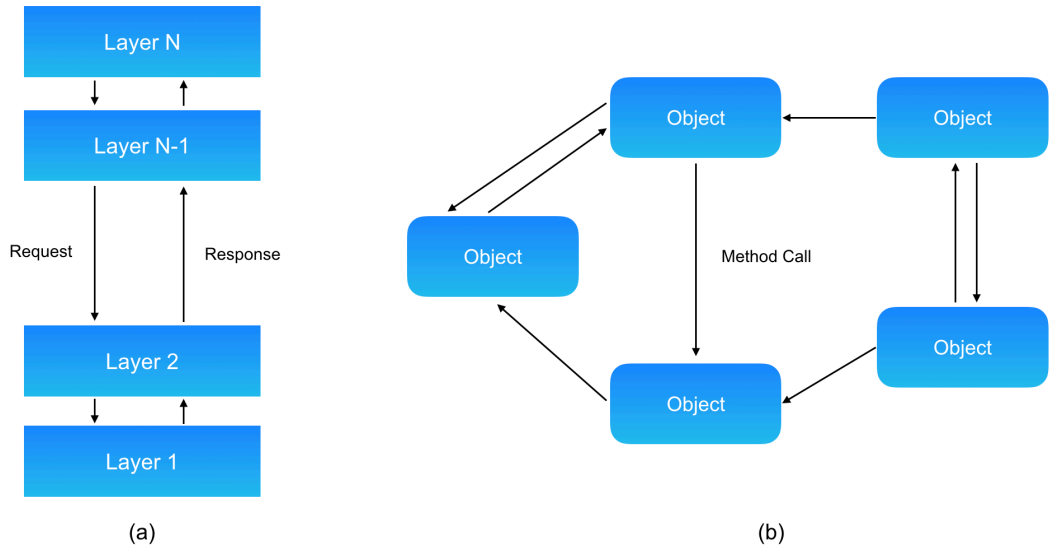


Figure 2.2.10 The layered and object-based architectural styles

The development of data-centred architectures is based on the notion of processes communicating using a common repository regardless of whether it is passive or active. The importance of these architectures' distributed systems may be comparable to that of the layered and object-based architectures. For example, an extensive amount of networked applications have been generated which depend on a shared distributed file system wherein almost all communication is conducted using files. Similarly, Web-based distributed systems tend to be data-centric; that is, the communication of processes takes place using shared Web-based data services.

Processes in event-based architectures usually communicate using the propagation of events that may also include data, as illustrated in Figure 2.2.11. Event propagation concerning distributed systems is often connected with publish/subscribe systems [75]. This means that once processes publish events, the middleware is responsible for ensuring that only the processes subscribed to those particular events receive them. Processes being loosely coupled is major benefit of event-based systems. They do not have to refer to each other explicitly, which is also called being decoupled in space or referentially decoupled.

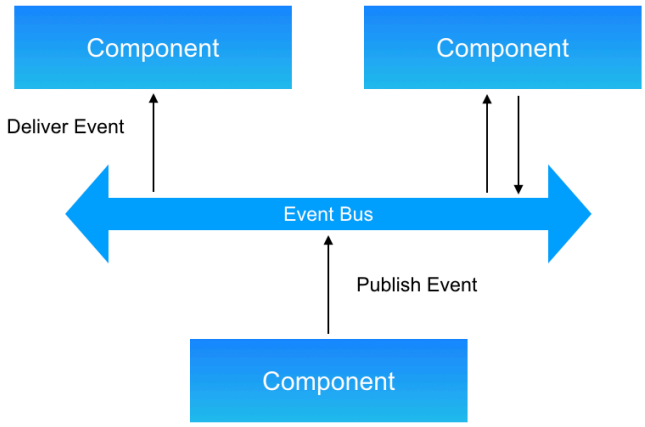


Figure 2.2.11 The event-based architectural style

It is possible to combine event-based architectures with data-centred architectures, resulting in shared data spaces. In shared data spaces, processes end to essentially be decoupled in time; that is, they require not both to be active when communicating. Moreover, several shared data spaces implement a SQL-like interface regarding the shared repository such that it is possible to access data through a description and not an explicit reference, similar to files.

Such software architectures are crucial for distributed systems because they all intend to attain distribution transparency to a reasonable extent. On the other hand, as previously stated, for distribution transparency, trade-offs must be made between aspects such as performance, ease-of-programming, and fault tolerance. Because no

particular solution exists which will cater to all possible distributed applications, researchers no longer focus on a single distributed system which can be employed for 90% of all possible cases.

2.2.4 Summary

There are numerous methods to organise distributed systems. Software architecture can be differentiated from system architecture. While software architecture focuses on the software's logical organisation such as how components interact, how they can be structured, and how they can be made independent, the system architecture is concerned with where the positioning throughout diverse machines of components forming a distributed system.

Distribution systems have several organisations. A vital class is one in which machines are classified into servers and clients. After a request is sent by a client to a server, the server produces a result which is returned to the client. Such client-server architecture indicates the traditional method of modularising software where a module establishes the functions that are available in a different module. Functions' natural physical distribution throughout a collection of machines can be attained by positioning various components on several machines.

Client-server architectures tend to be significantly centralised. It is often observed in decentralised architectures that the processes play an equal which form a distributed system, also called peer-to-peer systems. The organisation of processes in peer-to-peer systems forms an overlay network. This is a logical network where each process has a local list comprising other peers with which it can communicate. It is possible to structure the overlay network so that deterministic schemes may be deployed to route messages between processes. The peer list in unstructured networks tends to be random, which indicates that it is crucial to deploy search algorithms to locate data or other processes.

Moreover, architectural style is vital when examining architectures. Architectural style indicates the basic principle being followed when the interaction between the software components forming a distributed system is organised. Specifically, layering, event

orientation, data-space orientation and object orientation are four important architectures for distributed systems in the context of architectural styles.

2.3 Data Observatories

Data Observatories (DO) are virtual environments that include human–machine interface which offers an immersive experience while providing visualisations of extensive data sets as well as collaborative work. DOs are also called ‘cave’ referring to the University of Illinois’ Collaborative Automatic Virtual Environment (CAVE) project and because of their obscurity and an enveloping circular shape [2].

2.3.1 KPMG Data Observatory

The KPMG Data Observatory established in November 2015, is the largest DO in Europe. The Data Science Institute has designed, built, as well as housed the KPMG DO. This DO helps academics as well as the industry in visualising data so that they can discover new insights, while encouraging the use of a multi-dimensional and immersive environment for communicating complicated data sets and analysis [76]. Moreover, it aids decision makers in determining new implications as well as actions by examining data sets in a unique and innovative environment (Figure 2.3.1).

The DO offers three major modes. First is the Theatre Mode that uses the entire DO as one canvas. Second is the Decision Support Mode wherein the team is at the DO’s centre as various simulations or outputs can be seen around them, which enables the scenario to be run in real time and the data to be explored. Third is Hackathon Mode which involves five teams with every team working on one section of the five.

The hardware aspects are as follows [77]:

- 64 x 46” full HD Samsung UD46D-P Professional Video Wall Monitors with bezels of 3.5mm between two screens
- Powered by 32 rendering nodes
- Arranged as 4 rows and 16 columns

- Height = 2.53m
- Internal diameter = 6.00m
- 313-degree surround vision
- 16-point surround sound mounted on the top of each screen stack
- Total pixel count is 132,710,400 pixels (30,720 * 4320) which is believed to be the largest in Europe

Figure 2.3.1 The Data Observatory in Decision Support Mode

2.3.2 Other Data Observatories

Both medium-scale and small-scale multi-screen visualisation environments that have up to 4 screens have become popular among academics and the industry, whereas large-scale virtual environments remain rare as they pose extensive resource investment. It was in 1992 that the University of Illinois, Chicago, devised the first Collaborative Automatic Virtual Environment (CAVE) [78]. As shown in Figure 2.3.2, it presented images on screens that were organised in the shape of a cube.

The next generation of CAVE called CAVE2 [79] was developed in October 2012 and implemented actual LCD panels that were placed in a circle surrounding the team, as shown in Figure 2.3.3. In fact, this project led to the establishment of the KPMG Global Data Observatory, although it did create an in-house framework rather than depending on the API of CAVE, CAVELib [80].

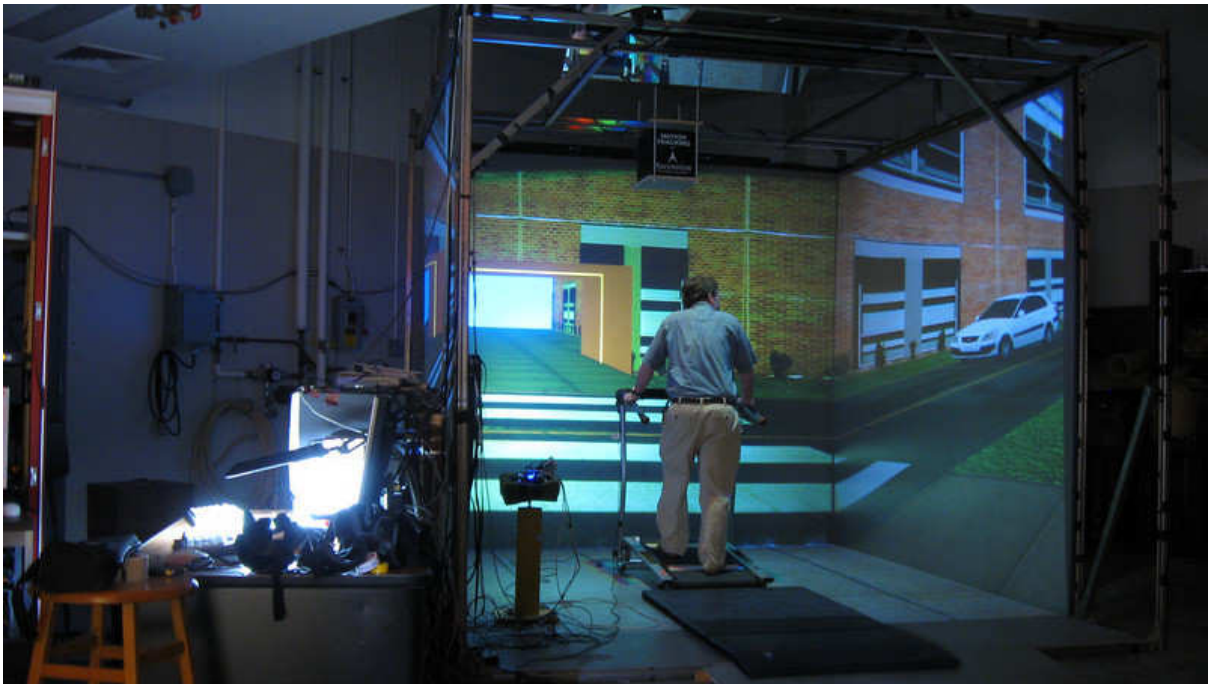


Figure 2.3.2 The first generation of CAVE system

Figure 2.3.3 The second generation of CAVE2 system

Further, within the industry, TechViz designed the TechViz Virtual Reality (VR) Showroom [81] that included a cave on the previously mentioned CAVE project's model, involving images being projected on screens organised in a complete cube shape, while emphasising on immersion in VR and 3D environments.

Chapter 3 Distributed D3 Framework Integrated with Data Observatory

3.1 Introduction

In Big Data era, there is an increasing need of analytic tools for obtaining insights from the growing large datasets is emerging. Visual perception as a primary tool of humans to retrieve information from the outside world, has the distinguish ability to rapidly differentiate patterns in a pre-attentive manner [26]. Visual analytics via data visualisation is therefore a very powerful tool in data analytics.

The Data Observatory is a state-of-art data visualisation facility that provides a scalable ultra-high-resolution displays in an immersive and multi-dimensional environment, which helps to uncover new insights and promotes the communication of complicated datasets for its users. The environment consists of 32 high-end graphical nodes with 64 high-resolution screens in total that creates a 313-degree immersive viewing arc for data observers and researchers [76].

In order to leverage the power of Data Observatory, we found that Data-Driven Documents (also known as D3.js) is a popular web-based data visualisation library that enables producing dynamic and interactive data visualisation in a variety of graphical forms. The standardised representation of D3.js improves the expressiveness and accessibility, and transforms offer large performance improvement and also enable transitions to be animated. Through performance benchmarks, D3.js is demonstrated to be at least two times faster than its ancestor [3].

However, due to the limits on the processing threads per browsing window in modern web browsers, the capacity of displaying large numbers of elements for a state-of-the-

art data visualisation application is limited. To address this bottleneck issue, we have designed a distributed framework that is based on D3.js, which is in hope to utilise the distributing power of the high-end graphical station cluster in Data Observatory that allows to serve a single or multiple data visualisations simultaneously.

3.2 Approach Comparisons

This section presents a comparison of various potentially helpful web technologies for determining the most appropriate approach in order to build the distributed framework. First, the underlying network communication protocols will be discussed and reviewed, followed by the possible web visualisation technologies, and finally a brief comparison between the preferred D3.js and other existing data visualisation libraries.

3.2.1 WebRTC vs. WebSocket

As a proposed web-based distributed visualisation system, the initial stage of framework building requires a highly efficient and robust communication method. This work begins by comparing the popular web communication plugin called WebSocket with the newly developed real-time peer-to-peer communication protocol called WebRTC [82].

The following observations were made after summarising their features and differences:

- WebSocket is a communication protocol that provides communication channels over a single TCP connection [83], whereas WebRTC is a free and open-source project that aids web browsers and mobile applications with real-time communication capability using simple APIs [82].

- WebSocket primarily focuses on allowing developers to deliver rich web applications, whereas WebRTC focuses on allowing users to establish peer-to-peer connections quickly and easily [84].

- WebRTC is designed to ensure high performance and high-quality communication of video, audio, and arbitrary data. It may, however, need a

signalling service for establishing the connection. WebSocket is designed to ensure that bi-directional communication takes place between the client and the server. While audio and video can be streamed over WebSocket, the service may not be as robust as WebRTC [84].

- WebSocket is developed in Java, JMS, and C++, whereas WebRTC is developed in JavaScript and HTML. Regarding scalability, WebSocket requires a server per session, whereas WebRTC can be peer-to-peer based.

- Compared to WebSockect, WebRTC has less security concerns concerning common security issues such as denial of service (DoS), man-in-the-middle, cross-site scripting, and client-to-server masking.

- Although WebRTC and WebSocket are both popular communication protocols, WebRTC is more commonly used in real-time communication applications.

- While being developed, WebRTC is relatively new and may only be available on certain browsers, whereas WebSocket is well supported by the majority of the browsers.

Following the comparison of the protocols above, WebRTC was implemented as the main network communication protocol. This is because of the potentially better performance as well as quality of the real-time communications among rendering peer nodes. Besides, WebRTC's peer-to-peer based network structure can further promote the scalability and flexibility of the potential distributed framework.

3.2.2 WebGL vs. HTML5

It is important to find a reasonable web-based data visualisation graphical presentation library or engine on which to build a favourable network communication solution. This

leads to comparing three relatively new yet very promising web technologies which are WebGL [85], HTML5 SVG [86], and Canvas [87]. Following are the main differences and features observed among these three:

HTML5 SVG

- Resolution independent
- Support for event handlers
- Best suited for applications with large rendering areas
- Slow rendering if the visualisation is complex
- Not suited for game applications

HTML5 Canvas

- Resolution dependent
- No support for event handlers
- Poor text rendering capabilities
- Faster rendering in complex visualisation
- Well-suited for graphic-intensive games

WebGL

- Resolution dependent (as enclosed in <*canvas*>)
- No support for event handlers (same as canvas)
- Best rendering performance
- Based on OpenGL ES 2.0 [88]
- Good support for 3D game applications

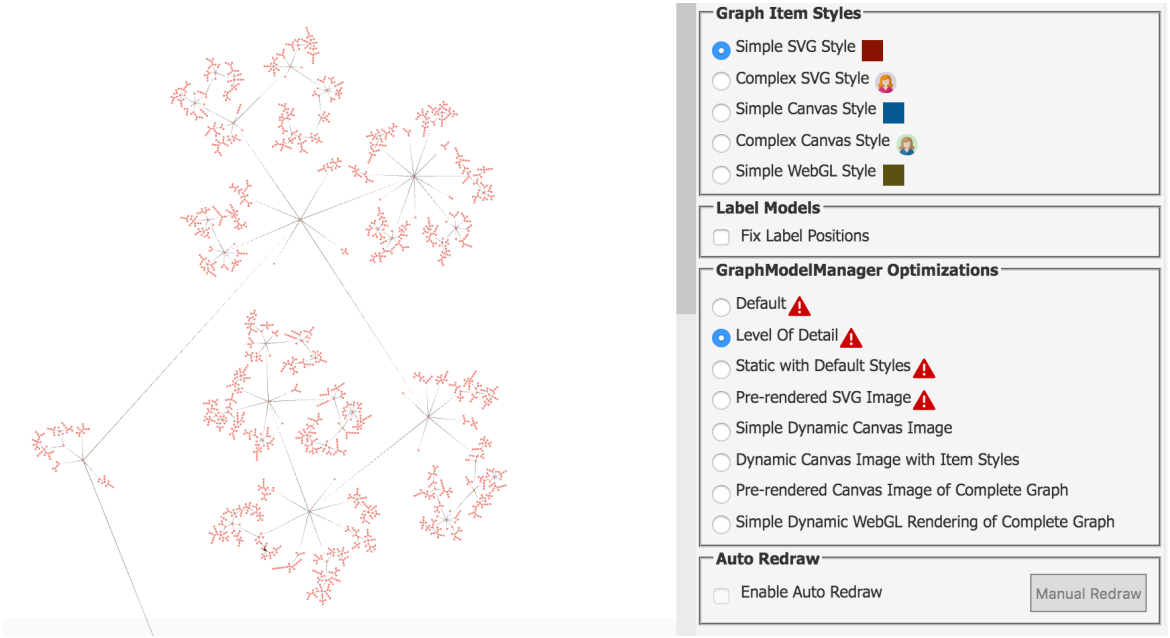


Figure 3.2.1 A test experiment by comparing the performance in frame rate (FPS) between HTML SVG, Canvas and WebGL

MacBook Pro 13" 2015	HTML5 SVG	HTML5 Canvas	WebGL
N = 100	60 (FPS)	60	60
N = 1000	38	56	60
N = 2000	20	28	44
N = 5000	N/A	11	20
N = 10000	N/A	N/A	12

Table 3.2.1 The test experiment result of HTML SVG, Canvas and WebGL

Meanwhile, we have also found and conducted a well-designed public-available test experiment by comparing these techniques with a different number (*N*) of animated rendering elements in frame rate (FPS) on a laptop as shown in Figure 3.2.1 [89]. The test results are consistent with the summary above, where WebGL has the best rendering performance among them, and HTML5 Canvas can render slightly faster than SVG as we can see in Table 3.2.1.

By considering all aspects of them, we consider that HTML5 SVG remains a good candidate, particularly considering its feature of resolution independence, although its performance is less satisfactory regarding more animated elements as shown in Table 3.2.1. This performance issue, however, can be improved by implementing it in the distributed framework. Moreover, although HTML5 Canvas showed relatively better performance, it has limited access to the event handling than SVG. Hence, it can be used along with SVG in a visualisation with less animation and more elements. In addition, WebGL is relatively new when starting the development, and although it has the same limitations concerning resolution and event handling as Canvas, it has significant potential regarding future development in terms of its rendering capacity and performance.

3.2.3 D3.js vs. Other Libraries

Following the aim of investigating approaches for building the distributed framework, the final step is to find whether or not existing graphical libraries are available and compatible with the previous preferences. Although it is known that D3.js is a strong candidate for this framework, this section aims to identify and compare all possible alternatives before committing to use one of them for developing the framework.

Library	Flexibility	Technology	Type of Charts
D3.js [90]	Large control via rich APIs	SVG	All
Infovis [91]	Large control via rich APIs	WebGL	All
Google Viz API [92]	Large choice in customisable charts	SVG	All
Springy.js [93]	Specialised charts	Raphael.js	Force-directed graphs

Polymaps.js [94]	Specialised charts	SVG	Maps
Dimple.js [95]	Same as D3, but more user-friendly in chart creations	D3.js	Axis-based charts
Sigma.js [96]	Specialised charts	WebGL	Line graphs
Raphael.js [97]	Large control via rich APIs	SVG, VML	All
gRaphael [98]	Pre-made charts for Raphael.js	Raphael.js	All
Leaflet [99]	Specialised charts	SVG	Maps

Table 3.2.2 A comparison of the selected web-based data visualisation libraries

Table 3.2.2 presents the findings of the most helpful libraries in this category. Primarily, these candidates' flexibility, underlying web technology, and the supported chart types were taken into account for summarising their main issues compared to D3.js with additional candidates as following,

- *Less Flexible*: Libraries such as *Data-wrapper* [100] or *Flot* [101] focus on rendering simple common charts and leave little room for visualisation customisation.
- *Based on D3.js*: D3.js is a popular tool that is often used as a low-level layer for higher-level libraries such as *NVD3.js* [102] which provides a collection of off-the-shelf visualisations.
- *Too Specialised*: Some libraries tackle one specific type of visualisation, although they can be added for specialised use cases, such as *Three.js* [103] can be used for rendering 3D visualisations.

Following the comparisons, it was decided to implement the original preference of D3.js. This is due to D3.js is developed based on HTML5 SVG, which has high scalability on scalable high-resolution screens that can be a good fit for the Data Observatory environment. Moreover, D3.js provides good support for building the highly interactive

state-of-art data visualisations regarding some of its built-in features. D3.js also has excellent extensibility as it is an open-source project and can thus allow developers to build customised plugins that can extend its functionalities.

3.3 Distributed D3 Framework Design

The design of the Distributed D3 framework is illustrated and discussed in this section. The concepts of distributed rendering and distributed data as the main features are detailed in the following subsections of §3.3.1 and §3.3.2. At the end of this section, the overall structure of the framework will also be discussed in §3.3.3.

3.3.1 Distributed Rendering

As one of the main features in Distributed D3, distributed rendering is designed to maximise the advantages of utilising distributed graphical computing power for data visualisations. It is realised by dividing a large visualisation rendering task into smaller pieces for each underlying distributed rendering node to undertake. Each node and its screens then only need to render and display their own margined and responsible parts of an entire visualisation.

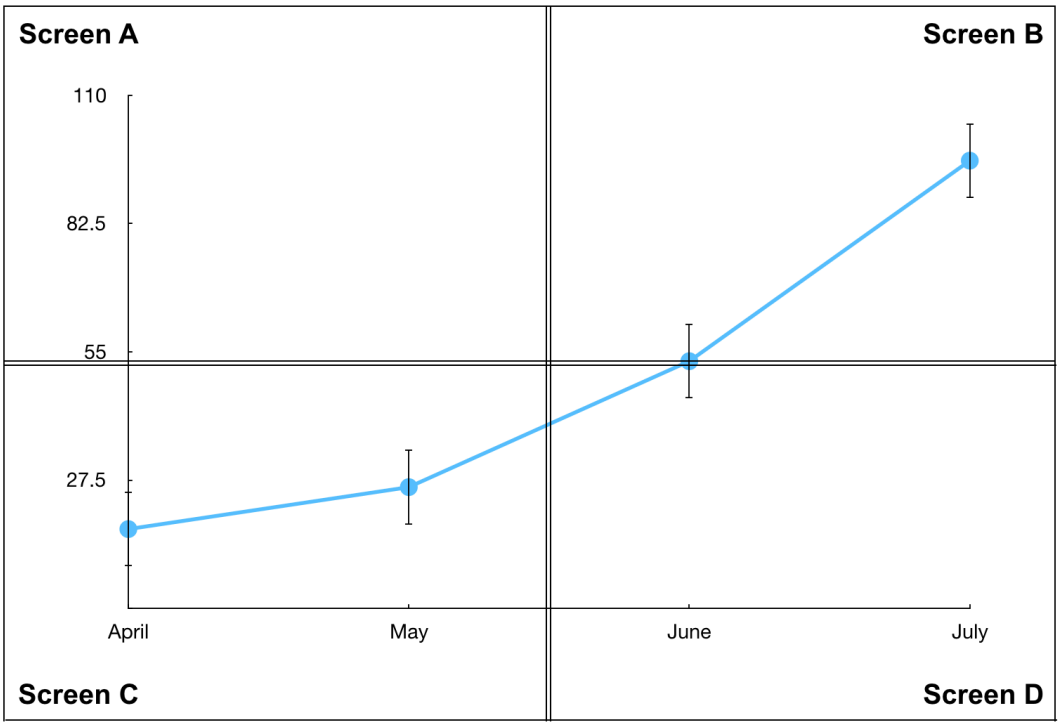


Figure 3.3.1 The illustration of a scatter plot with data line in the distributed rendering

As we can see in Figure 3.3.1, four scattering points are mainly distributed and displayed in screen B, C and D. It is clear that screen C and B will be in charge of the first two (April, May) and the last (July) data points respectively [104]. However, we notice the third data point of June is shared by screen B and D. To address this shared element issue, we will replicate the shared element in all contained screens at the initial stage of a visualisation. The same concept of sharing elements also applies to the *x* and *y* axes in this visualisation, which means both screen A and C will render a complete *x* axis in this case.

Apart from the static rendering at the initial stage of a visualisation, we also need to consider the case of rendering shared elements that involves animations and transitions, where the animated shared elements would only appear in an animation at certain screens. We address this dynamic rendering issue by predicting the potential recipient screens in a transition path, and then replicate and send this transition information with relevant elements to those screens via the real-time peer network channels. After that, a synchronising message will be broadcasted to all corresponding screens to ensure the transitions are started simultaneously, which is designed to improve the coherence of the cross-screen animations. We will further detail the implementations of static and dynamic rendering in the later section of §3.4.1.

3.3.2 Distributed Data

The mechanism of the distributed rendering enables the possibility of fetching data in a distributed manner. Since a distributed node may only require the subset of a dataset in order to render its margined part, it is therefore possible to predict and only fetch such subset of data from database. The main reason behind this design is that the size and complexity of a dataset which is stored in DOM tree often have a noticeable correlation on the performance of a web-based visualisation task.

When applying the concept of distributed data back to the illustration in Figure 3.3.1, it is clear that screen C needs to load the first and second data points, and thus screen B has the fourth data point. The third data point as a shared element needs to be pre-loaded in both screen B and D. However, we understand the extra data points need to be loaded under certain conditions, such as to draw the data line between data points in Figure 3.3.1. We therefore have allowed the data loading functions to fetch a given number of extra data points to extend its usability for certain needs.

As we know the shared *x* axis needs to be generated in both screen A and C, in order to avoid loading unnecessary data points in a screen just for the purpose of generating an axis, such as the case of screen A, we have also designed the data dimension function for obtaining the relevant scale information of a dataset. The implementations of data dimension and data loading functions will be detailed in the section of §3.4.2.

Moreover, similar to the design of dynamic rendering in the distributed rendering, a subset of the data may also need to be replicated and sent together with the transition information during a cross-screen animation. For the purpose of maintaining the size and complexity of the DOM tree, unnecessary data are designed to be dynamically removed to reduce the potential impact on the performance due to the replications.

3.3.3 Overall Structure

In order to realise the main distributed features of the framework design that we have discussed in the last sections, we have sketched the overall structure of Distributed D3 framework as shown in Figure 3.3.2 [104]. In general, the framework is designed to contain three main layers, which include the rendering layer, the data-accessing layer and the network layer.

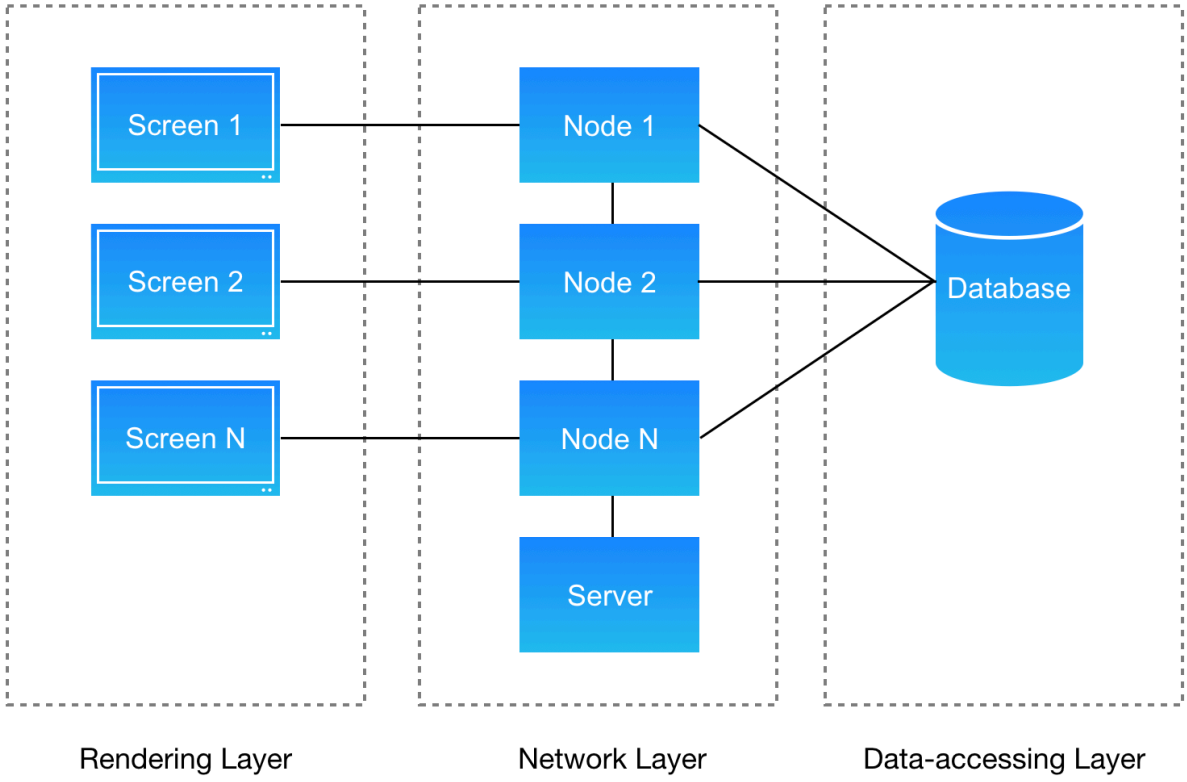


Figure 3.3.2 The overall structural design of the Distributed D3 framework

The rendering layer and data-accessing layer are the implementations of the distributed rendering and distributed data design. The network layer has the responsibility to establish the connections between a server and peer nodes, and also to build the fully connected network within peer nodes. The implementation details of these layers will be discussed in the sections of §3.4.1 to §3.4.3.

3.4 Distributed D3 Framework Implementation

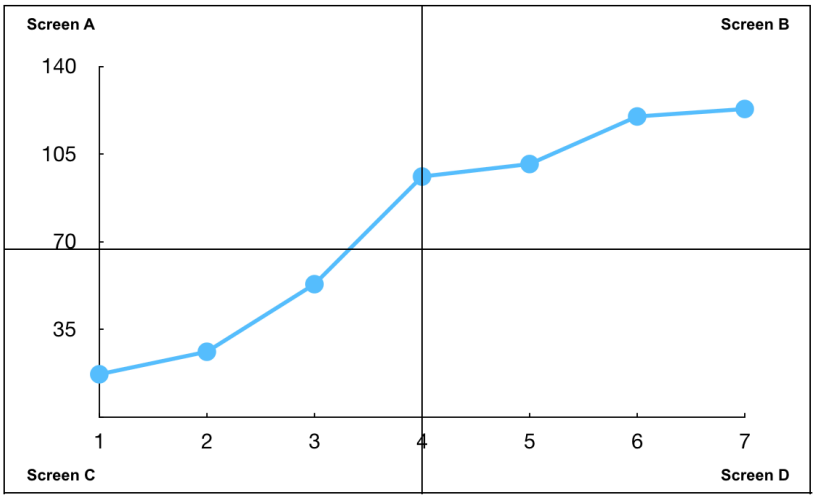
3.4.1 The Rendering Layer

The distributed rendering design can be fulfilled by creating following two rendering methods, which are static rendering by margining and dynamic rendering by transmitting. The implementation details of these two methods will be discussed in the following subsections.

Static rendering by margining

In the method of static rendering by margining, the rendering functions use the margins of individual screens to pre-determine the separated rendering tasks for each rendering node. Each node will then be able to render its margined part with additional shared elements at the initial stage of a visualisation.

At implementation level, the static rendering can be simplified to be relying on dividing data values into their corresponding screens. Since D3.js is essentially a data-driven library, once the library receives the data that are instructed how to render and display in a certain screen, it will then be able to generate a partial view of the visualisation in that screen.



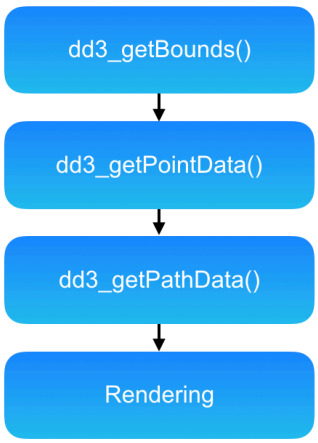


Figure 3.4.1 The illustration of the static rendering method by obtaining the margins (limits) of the dataset for each screen

Taking the scatter plot example in Figure 3.4.1, the static rendering method first uses dd3_getBounds function to obtain the margins (i.e. upper and lower limits) of the dataset in both *x* and *y* axis for each screen [104]. The corresponding data loading functions, in which case dd3_getPointData and dd3_getPathData for the data points and data line, will then be able to fetch a subset of data based on the data types. Hence, the rendering can be started afterwards by D3.js. Note that data loading functions will be further discussed in the section of §3.4.2 when we detail the data-accessing layer.

In detail, the dd3_getBounds method returns the upper and lower data limits for a certain browser window by using the domain and range ratio of data in that local browser window, as we can find the pseudocode in Figure 3.4.2, where the method will be divided as the first and second part [104].

In the first part of getBounds method in Figure 3.4.2, we are focused on initialising and preparing the domain and range for *x* and *y* axis. Specifically, if *scaleX* and *scaleY* have been defined from inputs, the method initialises the corresponding domains and ranges. Otherwise, data keys need to be given from inputs (i.e. *xKey* and *yKey*) in order to initialise the domains and ranges. These defined domains and ranges are checked afterwards to ensure their data sequences are in an incremental order, the order will be reversed if this is not the case, and an inverted flag is set to indicate the change.

After that, the second part in Figure 3.4.2 further defines the maximum and minimum values of ranges in *x* and *y* axis, where the values are given by the browser's margins if not limited by the data's ranges. Now we can obtain the corresponding maximum and minimum limits of domains by using the ratio between the domains and ranges, and finally the output limit can be returned and obtained.

Algorithm 1: The method of getBounds - Part 1

Input : data, scaleX, scaleY, xKey, yKey
Output: limit

```
/* Obtain data dimensions from data object                    */
d ← data.dataDimensions;
/* Initialise domains and ranges from inputs                  */
if scaleX is existed then
    domainX ← scaleX.domain().slice();
    rangeX ← scaleX.range().slice();
else
    domainX ← [d[xKey].min, d[xKey].max];
    rangeX ← [0, cave.svgWidth];
end
if scaleY is existed then
    domainY ← scaleY.domain().slice();
    rangeY ← scaleY.range().slice();
else
    domainY ← [d[yKey].min, d[yKey].max];
    rangeY ← [cave.svgHeight, 0];
end
/* Reverse domains and ranges if not in order                 */
invX ← 1, invY ← 1;
if domainX[0] > domainX[1] then
    domainX.reverse();
    invX ← −1;
end
if rangeX[0] > rangeX[1] then
    rangeX.reverse();
    invX ← −1;
end
if domainY[0] > domainY[1] then
    domainY.reverse();
    invY ← −1;
end
if rangeY[0] > rangeY[1] then
    rangeY.reverse();
    invY ← −1;
end
/* To be continued in Part 2                                  */
```

Algorithm 2: The method of getBounds - Part 2

Input : data, scaleX, scaleY, xKey, yKey
Output: limit

```
/* Define maximum and minimum values in X and Y          */
```
$limit \leftarrow \{\}$;
$minX \leftarrow Math.max(slsg.left(0), rangeX[0])$;
$maxX \leftarrow Math.min(slsg.left(browser.svgWidth), rangeX[1])$;
$minY \leftarrow Math.max(slsg.top(0), rangeY[0])$;
$maxY \leftarrow Math.min(slsg.top(browser.svgHeight, rangeY[1])$;

```
/* Obtain range's limits from domains by ratio           */
```
if $invX > 0$ **then**
  $limit.xmin \leftarrow domainX[0] + (minX - rangeX[0])/(rangeX[1] - rangeX[0]) * (domainX[1] - domainX[0])$;
  $limit.xmax \leftarrow domainX[0] + (maxX - rangeX[0])/(rangeX[1] - rangeX[0]) * (domainX[1] - domainX[0])$;
else
  $limit.xmin \leftarrow domainX[0] + (rangeX[1] - maxX)/(rangeX[1] - rangeX[0]) * (domainX[1] - domainX[0])$;
  $limit.xmax \leftarrow domainX[0] + (rangeX[1] - minX)/(rangeX[1] - rangeX[0]) * (domainX[1] - domainX[0])$;
end
if $invY > 0$ **then**
  $limit.ymin \leftarrow domainY[0] + (minY - rangeY[0])/(rangeY[1] - rangeY[0]) * (domainY[1] - domainY[0])$;
  $limit.ymax \leftarrow domainY[0] + (maxY - rangeY[0])/(rangeY[1] - rangeY[0]) * (domainY[1] - domainY[0])$;
else
  $limit.ymin \leftarrow domainY[0] + (rangeY[1] - maxY)/(rangeY[1] - rangeY[0]) * (domainY[1] - domainY[0])$;
  $limit.ymax \leftarrow domainY[0] + (rangeY[1] - minY)/(rangeY[1] - rangeY[0]) * (domainY[1] - domainY[0])$;
end
return limit;

Figure 3.4.2 The algorithm of the getBounds function with pseudo commands

Dynamic rendering by transmitting

The dynamic rendering by transmitting method is created to allow the distributed rendering in the case of animations and interactions. By watching the changes of transitions in an animated visualisation, the dynamic rendering functions will predict the end positions and calculate relative coordinates of a cross-screen transition, and then send relevant rendering information to the corresponding screens to prepare the animations. The animations will then be played and synchronised by a broadcasted message.

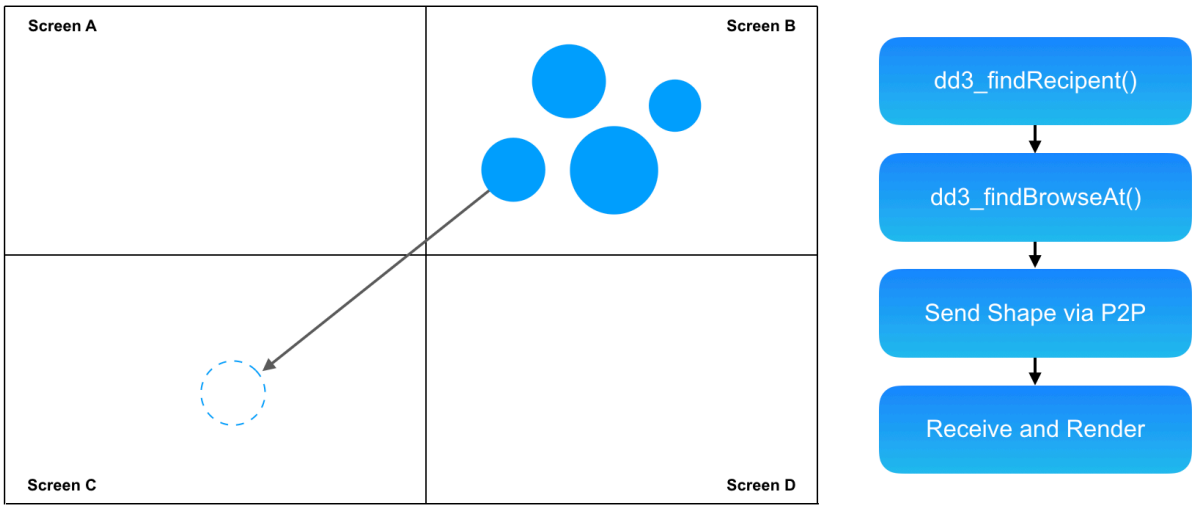


Figure 3.4.3 The illustration of the dynamic rendering by sending and receiving the shapes via real-time peer-to-peer network

An illustrative example of the dynamic rendering method can be seen in Figure 3.4.3, suppose we have several circular elements and one of them is moving from screen B to C, the method first attempts to find all cross-over screens as the recipients of that circular element by using dd3_findRecipients function, in which case they are screen B, A, D and C in order [104]. It then uses dd3_findBrowserAt function to obtain the relative transition paths and coordinates for each screen, in order to visually display the moving circular element as a continuous animation in a single large screen. Once these coordinates are obtained, it packs the transition information and forward it to the

corresponding screens via peer-to-peer network. The transition animations will be rendered and played on those screens after receiving and unpacking the information while taking account of the potential delay in the network transmission. Meanwhile, a synchronisation message will be broadcasted to improve the animation coherence across screens.

Algorithm 1: The method of findRecipients

```
Input  : element
Output: recipients[]
Initilisation:
recipients = [];
rect = element.getBoundingClientRect();
/* Check if rect is outside the current browser margin    */
if rect.top < 0 or rect.bottom > browser.height or rect.left < 0 or
 rect.right > browser.width then
    topLeftBrowser = findBrowserAt(hlhg.left(rect.left),
     hlhg.left(rect.top), 'html');
    bottomRightBrowser = findBrowserAt(hlhg.left(rect.right),
     hlhg.left(rect.bottom), 'html');
end
/* Loop over to find all recipients between topLeft and
   bottomRight browsers                                   */
for i in maxR do
    for j in maxC do
        if i ≠ browser.row or j ≠ browser.column then
            recipients.push([i,j]);
        end
    end
end
return recipients;
```

Figure 3.4.4: The algorithm of findRecipients function with pseudo commands

Algorithm 2: The method of findBrowserAt

```
Input  : horizontal, vertical, context
Output: pos
if context == svg then
    left = sghg.left(left);
    top = sghg.top(top);
end
pos = [];
pos[0] = top / browser.height;
pos[1] = left / browser.width;
return pos;
```

Figure 3.4.5: The algorithm of findBrowserAt function with pseudo commands

Specifically, the mechanism of findRecipients method can be seen in Figure 3.4.4 [104]. The method uses a local JavaScript function called getBoundingClientRect [105] to obtain the local coordinates of a bounding rectangle, where the rectangle contains the target element. The returned coordinates will then be used to check if the rectangle is outside the current browser; if this is the case, the method further invokes dd3_findBrowserAt as shown in Figure 3.4.5 to find the top left and bottom right recipient browser in the coordinates of the column and row, where the target element may be able to reach [104]. Once we know the coordinates of the top left (as minimal) and bottom right (as maximum) recipient browsers, we will then be able to loop over the columns and rows from the minimal value to the maximal, in order to obtain a full list of the browser recipients that will contain and receive the target element.

3.4.2 The Data-accessing Layer

For the variety of data structures in the potential visualisations, we have chosen MongoDB to be the main database option in this implementation. MongoDB is classified as a non-relational (NoSQL) database and featured by storing data in flexible, JSON-like documents with schemata [106]. In addition, we chose to use Open Data (OData) protocol [107] to build and consume the RESTful APIs [108], as it has multiple existing implementations and libraries well written in JavaScript.

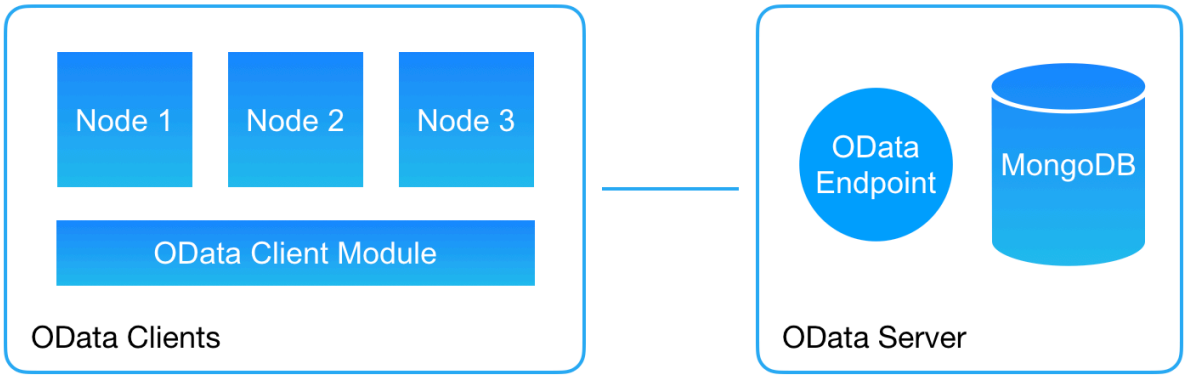


Figure 3.4.6 The structure of the data-accessing layer with OData APIs and MongoDB

The implementation of the data-accessing layer is illustrated in Figure 3.4.6, where multiple Distributed D3 instances on different nodes are able to access the OData endpoint via the OData client module [104]. On the other hand, the OData endpoint is built with MongoDB to allow querying database via the RESTful APIs. Such implementation improves the extensibility of deploying additional databases with the flexibility in other types for future development, and most of the common types of database are well supported by the OData endpoint framework.

In the aspects of data fetching and loading functions, the static rendering method defines the margined areas of data that will be rendered and displayed on the screens, this is further realised by selecting a subset of data from the original dataset in database. In practice, this distributed data fetching mechanism is useful for the coordinate-based data visualisation, such as scatter plots, bar charts and maps. However, for the visualisations

that are not based on coordinates usually require to define a customised filtering rule, otherwise the whole dataset will be obtained for each node, such as pie charts and treemaps. Fortunately, visualisations that are not coordinate-based generally have smaller size dataset. In Table 3.4.1, we summarise the data loading functions based on several common data types [104].

Data Types	Functions	Filtering Rules
Point Data	dd3_getPointData	Get the margins (i.e. upper and lower limits) from getBounds function
Path Data	dd3_getPathData	Get margins from getBounds with extended ranges for drawing the trend of a path
Bar Data	dd3_getBarData	Get the upper and lower limits based on orderingKey
Pie Data	dd3_getPieData	The filtering rule is not currently defined for pie chart due to the need of generating the pie shape
Undefined	dd3_getData	The filtering rule can be optional or customised on demand for any other undefined data types

Table 3.4.1 The list of data loading functions that are used to filter data by data types

Moreover, the data loading functions will further create the data queries to fetch the corresponding subset of data from database. The queries are handled by OData client module, where two fundamental data query functions – query() and queryWith() have implemented to interpret the basic and more advanced data queries based on the filtering rules.

```
http://config.host:config.port/serviceName/dataName?$select=da
ta.name&$orderby=data.order&$filter=data.filterRules
```

Figure 3.4.7 The main components of a typical OData query string with individual fields

In detail, a typical query string in OData can be seen in Figure 3.4.7, where the main purpose of query() and queryWith() method is to construct such a data query based on the given dataset and its specific filtering rules. As we can see the pseudocode in Figure 3.4.8, the basic query string consists of *$select*, *$orderby* and *$filter* in addition to the

host configurations [104]. The query() method then simply assembles the individual components and send this query string to the OData service endpoint via an OData client module - o.js [109] for the callback result. In comparison, an advanced data query can be assembled by queryWith() method, where additional modifiers are allowed, such as *$top* and *$skip*. Moreover, a custom parameter is also allowed in the advanced query which can be used to apply modifications on the returned data in callback.

Algorithm 1: The method of query and queryWith

Input : dataType, dataId, select, orderby, queryFilter, top, skip, custom
Output: callback

```
/* Constructing the query string header                    */
queryStr = odata.config.protocol + "://" + odata.config.host + ":" +
 odata.config.port + "/";
queryStr += dataId + "?";
/* Basic query in query() method                           */
queryStr += "$select=" + select;
queryStr += "&$orderby=" + orderby;
queryStr += "&$filter=" + queryFilter;
/* Advanced query in queryWith() method                    */
queryStr += "&$top=" + top;
queryStr += "&$skip=" + skip;
/* Querying via the OData client module                    */
oHandler = o(queryStr);
oHandler.get(function (data) {
    if custom then
    |   // modify return result
    end
    callback(dataType, dataId, data);
});
```

Figure 3.4.8 The algorithm of the query and queryWith method with an OData client module

3.4.3 The Network Layer

In the implementation of the network layer, we have chosen the newly developed WebRTC to be the main communication protocol in addition to the SignalR [110]. The WebRTC protocol is featured by allowing direct peer-to-peer communication without installing extra plugins, which is well supported by the most of up-to-date modern browsers [82]. To further wrap and implement the protocol, we found the open-sourced framework PeerJS which is an easy-to-use and configurable implementation that can be useful in the development [111].

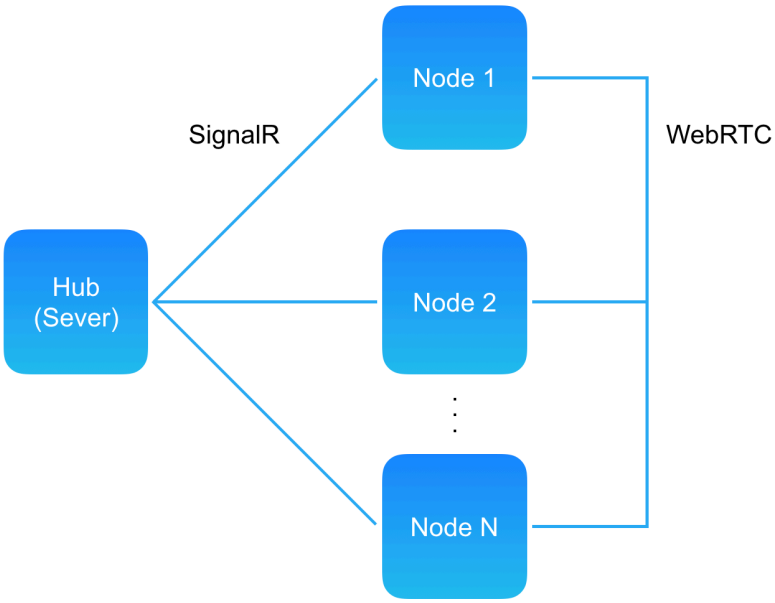


Figure 3.4.9 The underlying fully-connected peer network (with WebRTC) and the star-shape controlling network (with SignalR) in the network layer implementation

As we can see the network structure in Figure 3.4.9, we have implemented two types of underlying networks in this layer [104]. The peer network, which is fully connected with all peer nodes by WebRTC, is mainly designed to deal with the dynamic peer updates, including the events and communications in the animations and interactions. The controlling network is star-shaped and established by the SignalR hub (server), which is mainly responsible for broadcasting messages from the hub to peer nodes,

including commands or configurations. In practice, this separated controlling network is useful to improve the reliability of the framework, as it helps to fast detect a disconnected rendering node and also to better synchronise the animations by direct broadcasting.

When we implement these two integrated networks in detail, the framework is first set to establish the SignalR connections between the hub and peer nodes at the initialising stage, wherein the peer nodes connect to the hub via the SignalR server address so that they can obtain their unique connection IDs on the connections. After that, the nodes will attempt to connect to the peer server to get their individual peer IDs, and then send them back to the hub.

On the hub side, after checking the total connected number of clients are equal to the configured client number, it broadcasts a full list of the connected nodes with all of their connection and configuration information including peer IDs. The peer nodes will then be able to establish the fully connected peer network accordingly. The pseudocode of this entangled network initialising process can be seen in Figure 3.4.10 [104].

Algorithm 1: Network establishment on App Hub

```
/* Initialise variable and arrays                          */
var clientNum;
var connAry, infoAry;
/* Update conn array if receive a new connection           */
Procedure receiveNewConnection(conn)
    connAry.push(conn);
end
/* Update info array when all nodes connected              */
Procedure updateInformation(info)
    infoAry.push(info);
    if connAry.length == clientNum then
        broadcastInformation(infoAry);
    end
end
/* Broadcast info array to all connected nodes             */
Procedure broadcastInformation()
    connAry.broadcast(infoAry);
end
```

Algorithm 2: Network establishment on Peer Node

```
/* Initialise variable and array                           */
var nodeInfo, infoAry;
/* Initialise peer-to-peer network                         */
Procedure initialisePeerNet()
    peerConn = connectToPeerServer(peerServerAddress);
    nodeInfo.peerId = peerConn.peerId;
    waitingToBeConnected(function(){
    // waiting to be connected by peer
    });
end
/* Initialise signalR network                              */
Procedure initialiseSignalRNet()
    sigConn = connectToSignalRServer(signalrServerAddress);
    sigConn.updateInformation(nodeInfo);
    broadcastInformation = function(data){
        infoAry = data;
        foreach info in infoAry do
            peerConn.connectToThisPeer(info.peerId);
            // connect to all other peers
        end
    }
end
```

Figure 3.4.10 The pseudocode for establishing the fully-connected peer network via the established SignalR network

3.5 Results

In order to demonstrate and benchmark the implementation results, we have written a number of common chart examples for this integrated version of Distributed D3 on the Data Observatory. Meanwhile, we have also designed the benchmarking toolkit in order to evaluate the potentially improved scalability and performance of the framework in a different number of node configurations. We will demonstrate these chart examples in the section of §3.5.1 and then discuss the benchmarking results in §3.5.2.

3.5.1 Demonstrating Examples

After deploying the chart examples on the integrated Distributed D3, we set up a testing environment for the demonstration that includes 2 rendering nodes and 4 high-resolution screens on Data Observatory. In Figure 3.5.1, the screenshot shows the scatter plot chart example that has been successfully running on the testing environment in a distributed manner, where each screen only renders a portion of the entire visualisation [104].

This distributed mechanism can be confirmed by investigating the DOM structure in Figure 3.5.2, in which each browser window only holds a limited number of rendering elements that are essential to visually create the visualisation as one instance for distributed rendering [104]. Specifically, if we count the visually visible points on each screen, the result is matching the number of circle elements (as points) that are existed in DOM tree. Since the circle elements are essentially created by the corresponding subset of data in our design, the test can also confirm the distributed data has been successfully implemented in this example.

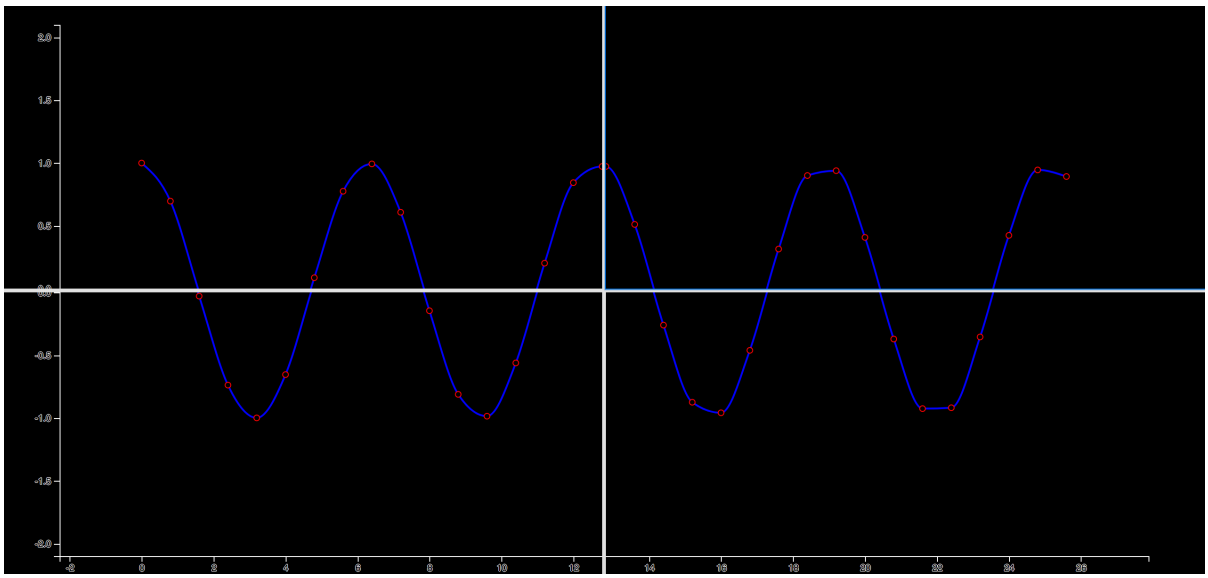


Figure 3.5.1 The demonstrating example of the scatter plot visualised by Distributed D3

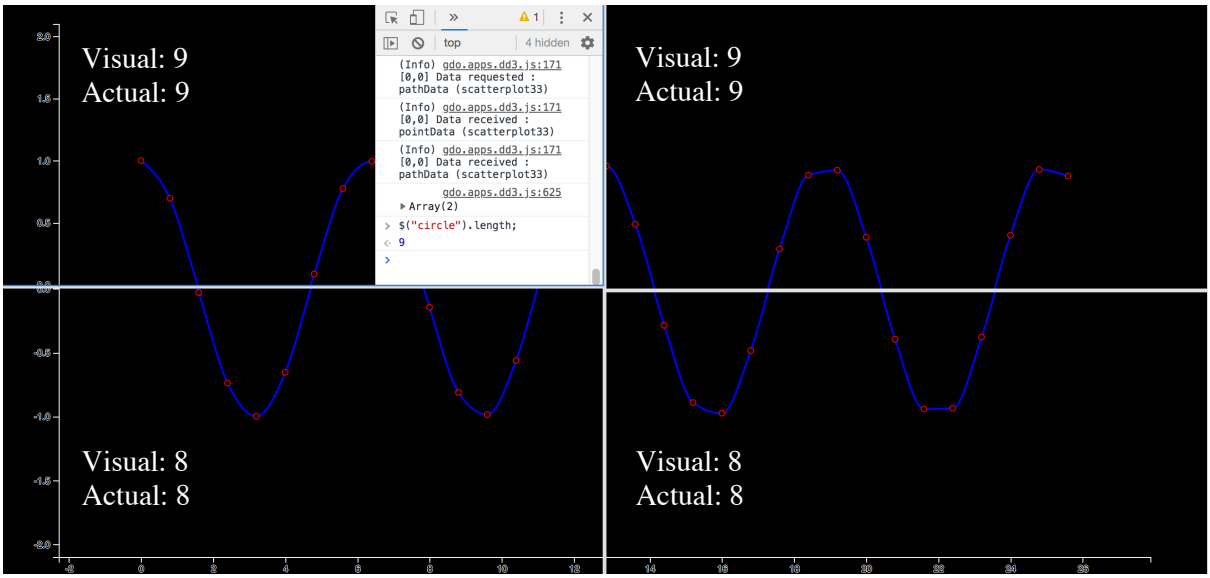


Figure 3.5.2 The inspecting results of counting circular data points in each screen

The bar chart and pie chart are also successfully deploying on the testing environment as we can see in Figure 3.5.3 and Figure 3.5.4 [104]. By inspecting the DOM structure, we can confirm the bar chart only requested and rendered a margined range of data for each screen, and it also properly generates *x* and *y* scales by only using the dimension information of its dataset. In comparison, the pie chart has requested the entire dataset

for each screen due to the needs of generating an entire pie shape, which also can be regarded as one large shared element from the view of our design philosophy in this version.

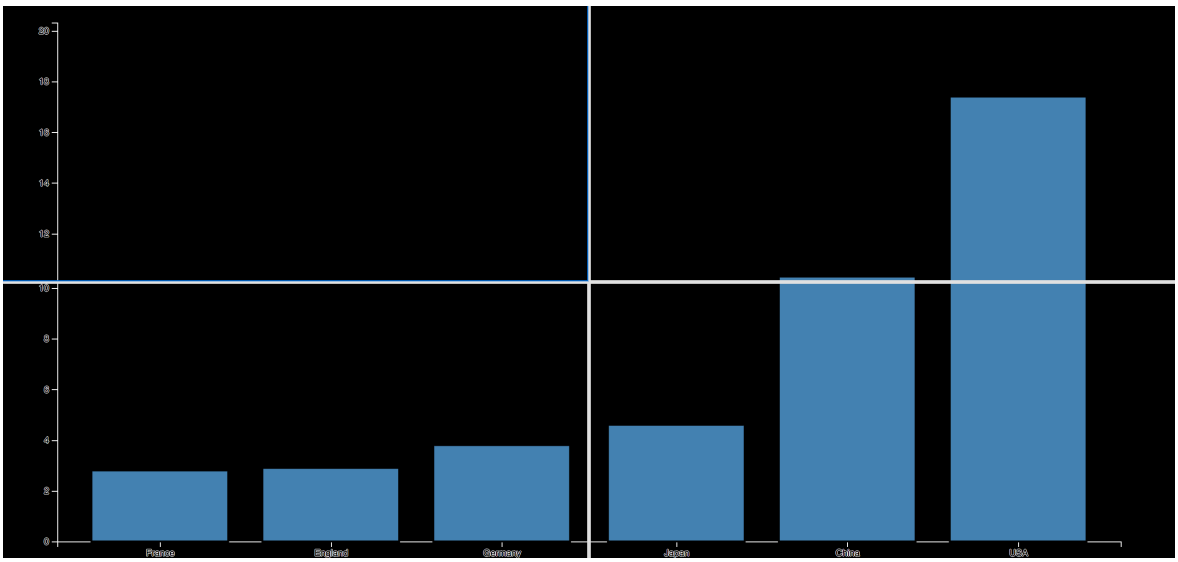


Figure 3.5.3 The demonstrating example of the bar chart that is visualised by Distributed D3

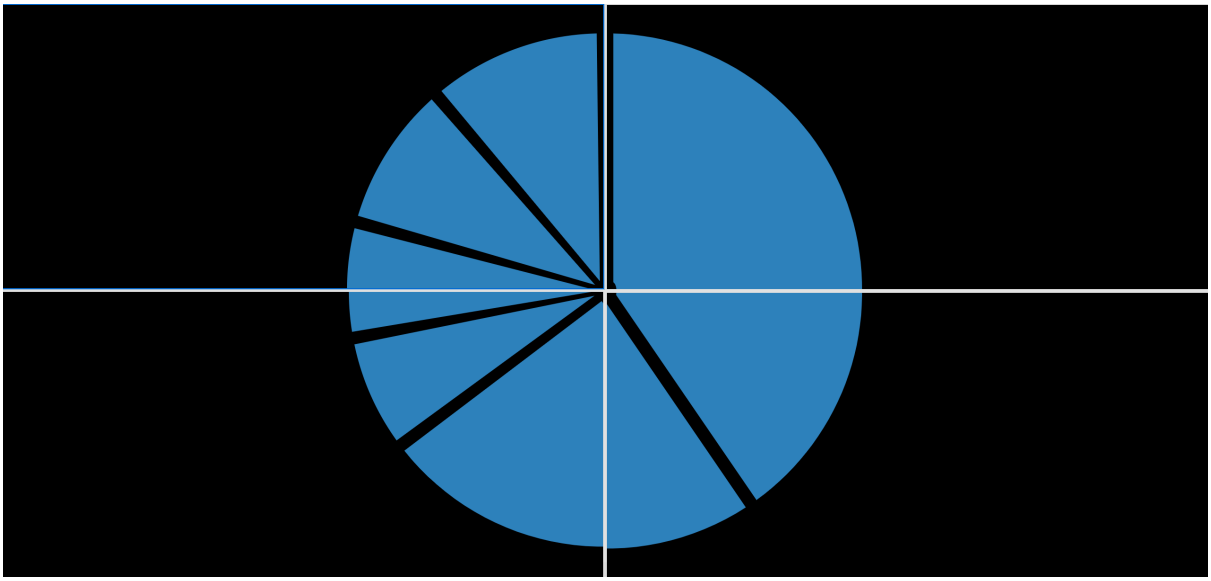


Figure 3.5.4 The demonstrating example of the pie chart that is visualised by Distributed D3

Moreover, these chart examples have also been tested on other numbers of screen configurations, and they have been working and running successfully. The framework

scalability and its performance will be more precisely evaluated and benchmarked in the next section.

In addition, a London Tube map was developed by a colleague in the Data Science Institute at the time of finishing this integrated version of Distributed D3, the demonstration in Figure 3.5.5 shows the animated popularity of the tube stations at the peak and off-peak time in London [112].

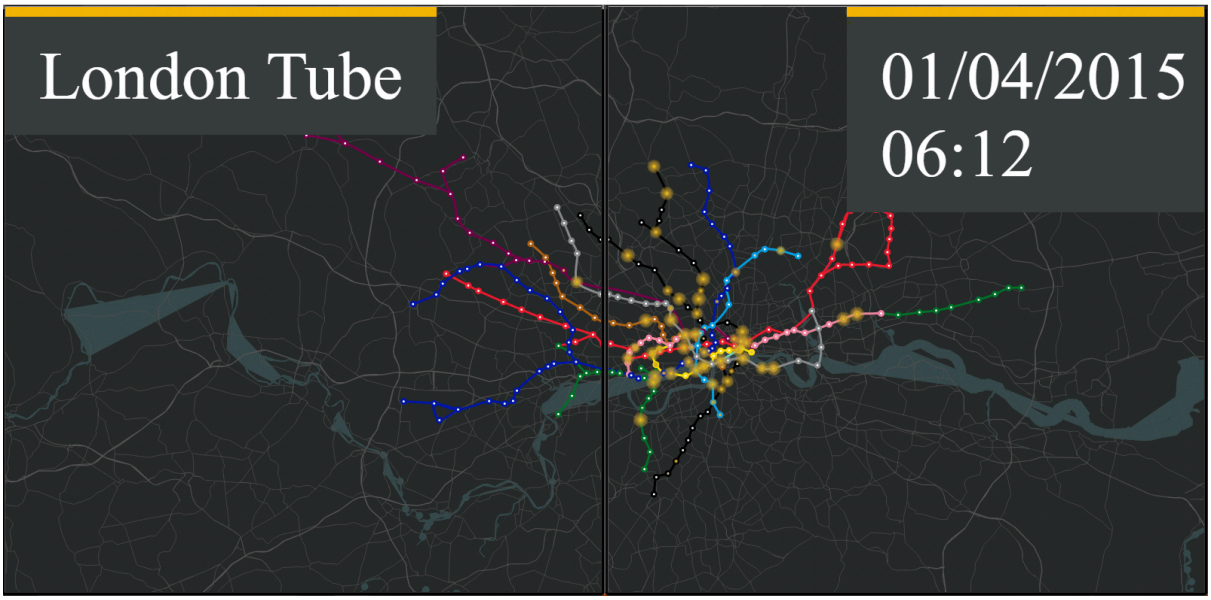


Figure 3.5.5 An demonstration of the London Tube map with animated entries and exits on the peak and off-peak time of the tube stations, which is built with Distributed D3

3.5.2 Performance Benchmarking

In order to benchmark the framework in all possible configurations on Data Observatory, we have designed and implemented the benchmarking toolkit that creates distributed random circles on the available screens in a test, and then it randomly moves the generated circles across screens. The number of animated circle elements and the average FPS during the moving animations will be recorded accordingly in different size of node configurations.

A screenshot of the implemented benchmarking toolkit for 2-screen test environment can be seen in Figure 3.5.6 and Figure 3.5.7, where we put the test environment under stress in the second screenshot [104]. The full benchmarking tests will include the configurations from 1 node (2 screens) to 32 nodes (64 screens) by doubling the node number in each step. The test result has been shown in Figure 3.5.8, in which the charts show the average FPS against the number of animated circle elements for different node configurations.

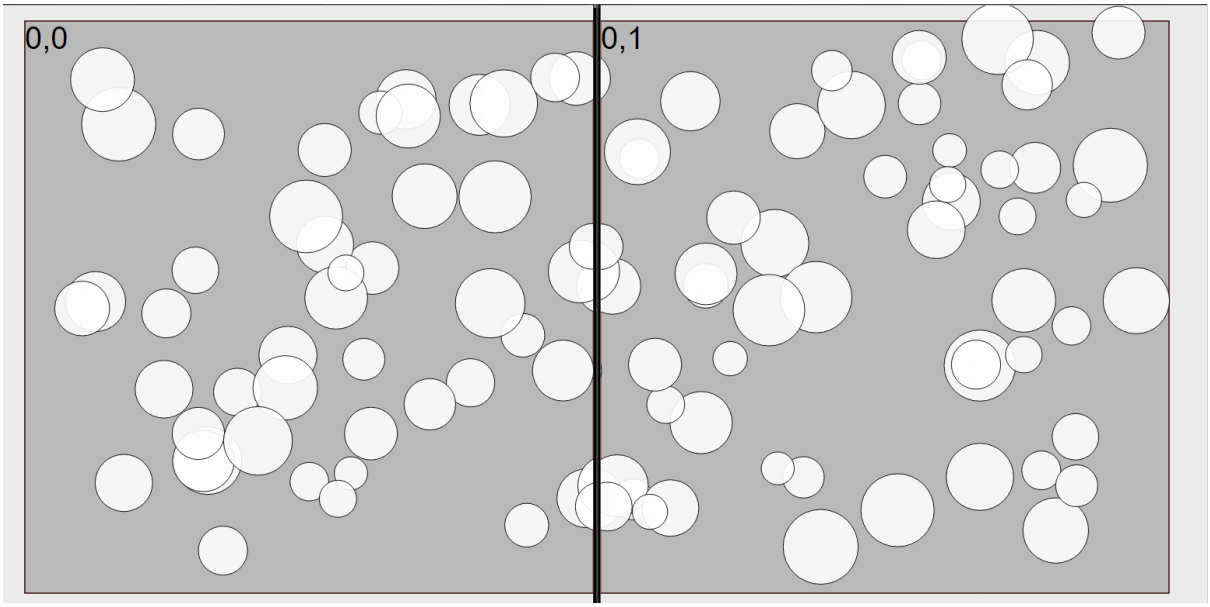


Figure 3.5.6 The screenshot of the benchmarking toolkit in test of 2-screen setting with 100 animated circle elements

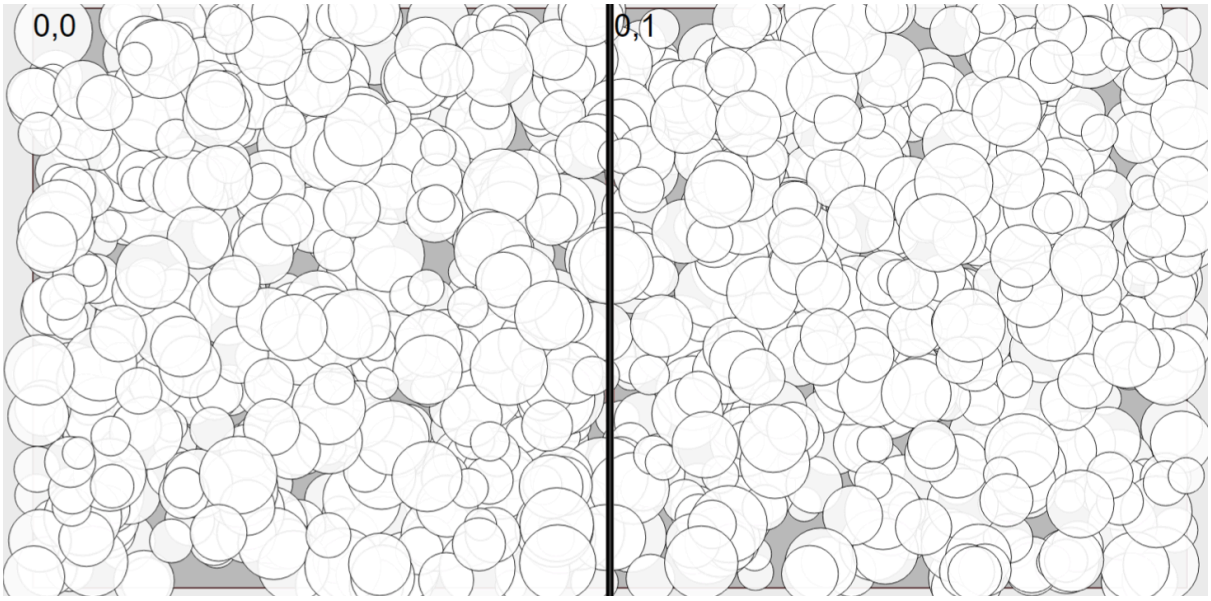


Figure 3.5.7 The screenshot of the benchmarking toolkit in the test of 2-screen setting with 1,000 animated circle elements

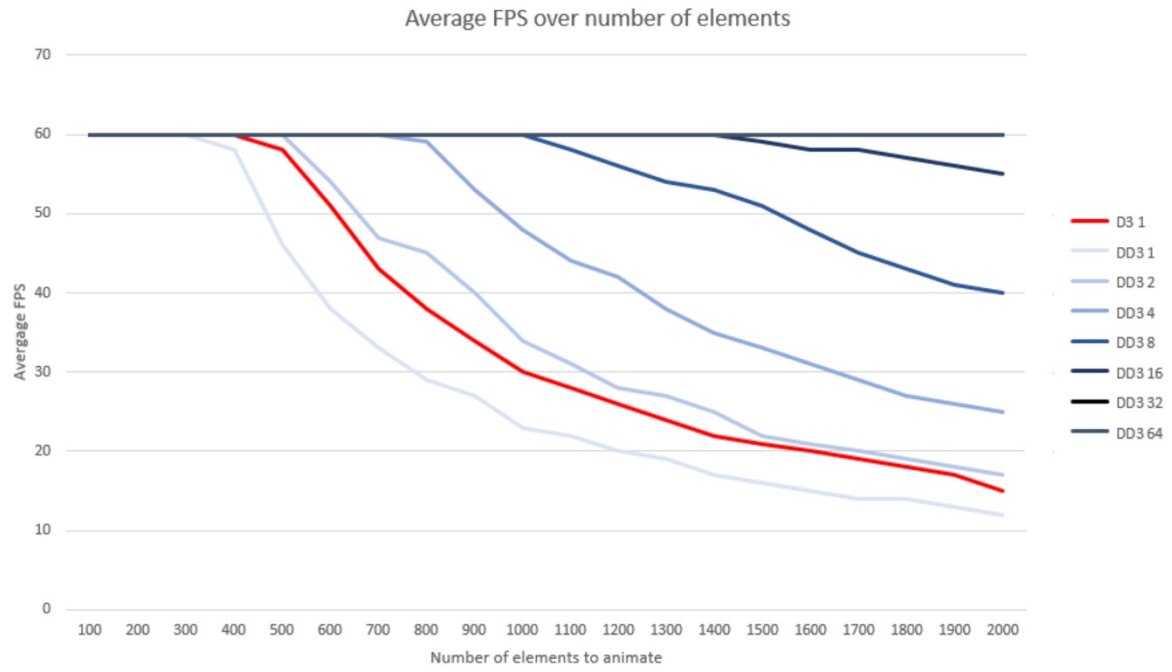


Figure 3.5.8 The benchmarking result of the integrated version of Distributed D3 in comparison with running D3 alone (in red)

In Figure 3.5.8, we notice that, with more available distributed rendering nodes, the visualisation can steadily provide a higher frame rate (FPS) with a larger number of animated elements [104]. D3.js alone has its acceptable animation smoothness (i.e. FPS > 24) [113] threshold at approximately 1200 elements and optimised animation threshold (FPS ≈ 60) at approximately 500 elements, which are regarded as the reference points in this test. The performance is slightly decreased with Distributed D3 with 1 node (1 screen) setting that is potentially caused by the overhead of applying distributed implementations on a single node. Whereas such disadvantage starts to be outweighed when the configuration changes to 1 node (2 screens) setting, since we know that an extra screen can provide an extra browser window with more available processing threads to a visualisation despite on a single node. The test results thus far demonstrate that the performance bottleneck of original D3 has been successfully overcome by utilising Distributed D3.

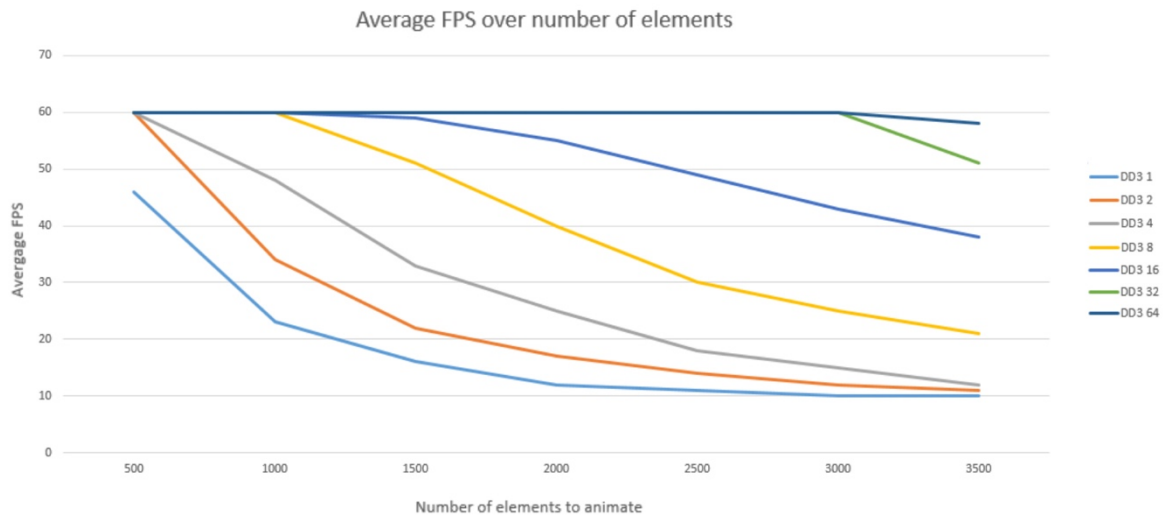


Figure 3.5.9 The benchmarking result of the integrated version of Distributed D3 for investigating the optimised animation threshold in all configurations

Besides, in order to find and compare the optimised animation thresholds for all configurations including 32 and 64 screens, we set the animated number of elements from the original D3's reference point of 500 elements to the last performance declining

point. As we can see the results in Figure 3.5.9, with the configuration of 32 and 64 screens in the test, we have been able to improve the performance bottleneck (for the optimised animation) of the original D3 from 500 animated elements to 3,000 and 3,500 elements respectively, in the development of this integrated version of Distributed D3 [104].

3.6 Discussion

As a complex distributed system, it is clear that the potential bottlenecks of Distributed D3 need to be identified in order to further improve the existing framework. Two main relevant areas are worth to be investigated that include the potentially large number of DOM interactions and possible network latency during transmitting. Meanwhile, we are interested in finding new approaches to further optimise the framework.

In addition, more demonstrating examples are expected to be developed on the current Distributed D3 for the purpose of providing basic visualisation code examples for more advanced visualisation applications. In the meantime, developing various examples can also help to find potential issues in the current version of Distributed D3, and hence to improve the stability of the framework.

Moreover, although the integration of current Distributed D3 framework provides simplicity to leverage the distributed graphical computing power in the Data Observatory, it is clear that an independent implementation of the framework will need to be designed in the later development for more generic usages. We would expect to enable Distributed D3 to be configured and used on a variety of visualisation environments, from a small cluster of computers to the modern data observatories.

3.7 Conclusion

In this chapter, we proposed and developed the integrated version of Distributed D3 framework to resolve the performance limits of running D3.js alone on Data Observatory. We illustrated and discussed the framework design and implementations after comparing a variety of possible designing approaches. In particular, the static and dynamic rendering methods are created in order to fulfil the feature of distributed rendering, a variety of data loading functions are developed for the feature of distributed data. In overall, three individual layers are designed for Distributed D3 including the rendering, data-accessing and network layers. The implementation result of the distributed framework is functioning as expected as we can see from the demonstrating examples. The benchmarking results further show the improvement of the overall performance and scalability comparing to the original D3.js. In addition, we have also discussed the potential issues in this integrated version of Distributed D3 and suggested possible solutions to further improve the framework in the next stage.

Chapter 4 Distributed D3 Framework Optimisation with a Demonstrating Application

4.1 Introduction

The recent development of the integrated version of Distributed D3 has been beneficial to the Data Observatory, in terms of including a powerful and scalable D3-based data visualisation framework to the existing visualisation system and library. Several remarkable examples have been developed by researchers and developers at Data Science Institute, including London and Shanghai Metro Maps [112].

However, due to the complexity of implementing a distributed framework in practice, we understand that it is essential to maintain the framework usability by continuously optimising its performance and creating new features to the current framework. We have noticed a number of emerging new techniques that might be worth to be implemented with Distributed D3 to further improve its performances on large-scale visualisations, such as an implementation of the concept of virtual DOM [114] - React.js, which is recently developed by Facebook [115].

In this chapter, we focus on optimising the previously developed integrated Distributed D3 by looking into the potential performance bottlenecks of the existing framework. We then test the possibility of implementing our findings before we propose modifications and improvements to the framework. After implementing these proposed optimisation approaches, we will benchmark the optimised framework to be compared with the previous development. Meanwhile, we will also design and implement an interactive demonstrating visualisation application on this new version. At the end of

the chapter, possible further improvements will be included and discussed for the future work and development.

4.2 Distributed D3 Framework Bottleneck Analysis

Identifying the bottlenecks of the current Distributed D3 framework is essential for the purpose of optimising the framework performance as a next step in the development. From the benchmarking results of the integrated framework in the last chapter, we have known the dropping frame rate (FPS) is correlated to the total number of animated elements that are rendered on the screens. The result makes common sense that increasing number of animated elements require more graphical computing resource to ensure the smoothness of the animation. However, in order to ascertain if it is still possible to enhance the framework performance under the same visualisation condition (i.e. the same visualisation environment), we will look into the underlying DOM interactions between the rendering functions and hosting browsers.

4.2.1 Excessive Garbage Collections

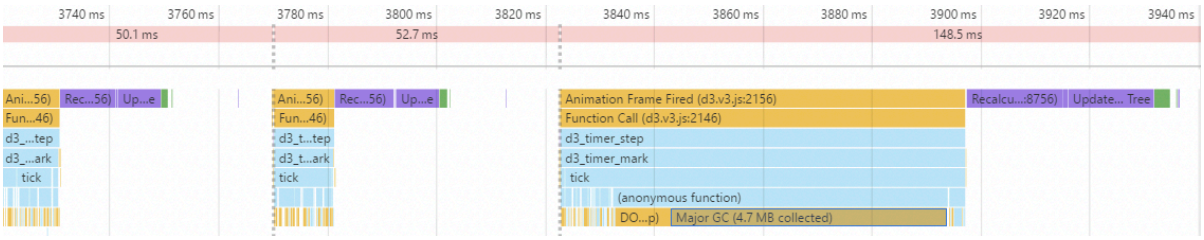


Figure 4.2.1 The Garbage Collection (GC) frame that is captured in the benchmarking test

The investigation of the framework performance bottleneck is started at the lagging point that we have observed in the benchmarking tests of the integrated Distribute D3 on Data Observatory, in which the benchmarking animation starts to have noticeable delay from the first frame to the next. We then look into this event by inspecting and recording the lagging point via the performance metric provided in Chrome DevTools [116]. As is shown in Figure 4.2.1, we notice an extra-long frame starts to appear during the animation at the lagging point, which is roughly 3 times (148.5ms) longer than an average normal frame (51.4ms) [117]; meanwhile, an extra step is appearing and lasting

in the lagging frames which is called Major GC [118]. According to the research paper published by Google [119], Major GC in a Chrome browser stands for major garbage collection of the whole memory heap that is performed if the size of live objects has exceeded a pre-defined limit in order to reduce the high memory usage. Normally, the garbage collection marking latency is related to the number of live objects that need to be marked; in the case of the whole heap, it can potentially take more 100ms for a large web page [119], which can be corresponding to the time lag we have observed.

Although the garbage collection is an essential step of dynamic memory management automated by Chrome V8 JavaScript engine [120], we also find the current version of D3 v3.5.6 does not manage the DOM access in an optimised manner, so that it can avoid triggering unnecessary major garbage collections in a further investigation. In particular, the ongoing animations would be paused if the DOM tree is unavailable or in an inactive state; and they should be resumed in sequence when the DOM becomes available again. However, the current version of D3 library resumes paused animations all at once in order to catch up the playtime, which can cause a sudden and massive workload on the DOM at the animation resuming stage; and it can further lead to excessive garbage collections if the visualisation is in large-scale that has high impact on the memory usage.

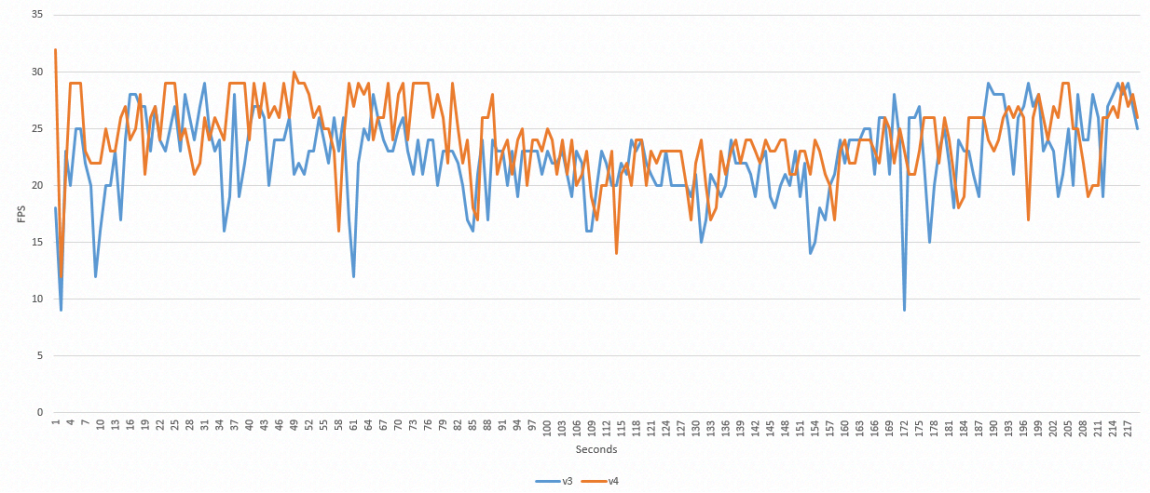


Figure 4.2.2 The comparison of the frame rates (FPS) between D3 v3.5.6 and v4.0.0

Since D3 has noticed this issue in the animation resuming event, it improves its timer and animation mechanism in the newer version of v4.0.0 [121]. In which case, the D3 timer will be frozen, and the animation will be held when the DOM is not available, and then both timer and animation will be resumed after the DOM becomes available again to avoid unintended animation effects and memory impact [122]. The improvement may only optimise the framework performance on a certain extent by avoiding triggering unnecessary garbage collections. By experimenting the benchmarking example on D3 v3.5.6 and v4.0.0, we can see a limited average performance improvement in Figure 4.2.2, wherein the average frame rates (FPS) over a time span of 220 seconds are 22.5 and 24.3 in v3.5.6 and v4.0.0, respectively [117]. Hence, a similar result would be expected when applying this newer version of D3 to the Distributed D3.

4.2.2 Massive DOM Interactions

By further inspecting the runtime JavaScript function calls on the DOM in browser, we notice that there is one remarkable function that has been heavily used in the benchmarking example, which is setAttribute() [123]. The function simply assigns an attribute to a specified DOM element as simple as it sounds to be; however, it can be a huge impact on the visualisation performance if the existing DOM tree contains thousands of elements that need to be set, such as in the test example.

If we abstract the issue from observation, imagining the animation of randomly moved circles, thousands of coordinate attributes need to be assigned to their corresponding DOM elements at a single animation frame, the overall action can lead to a noticeable drop of frame rate if it repeatedly triggers the garbage collections due to the high memory usage. Therefore, this performance limitation issue is not only the bottleneck of Distributed D3, but also a main performance constraint in the D3.js and other resource-intensive web-based visualisation frameworks.

To address this openly known performance bottleneck [124], we find that the virtual DOM is a newly emerging concept of managing and manipulating DOM tree structure with a reduced total number of DOM interactions; React is popular implementation with integrating this concept [114]. Essentially, it creates a virtual view of the current DOM tree structure in memory and then only applies the DOM structure changes when they are necessary [125]. The trade-off of this virtual approach is the potentially higher usage of the memory for stacking the virtual DOM; however, this may be less problematic with the high-end graphic-focused stations such as on the Data Observatory.

Although we understand this virtual approach may have its limitation on the animation-based performance optimisation due to the necessity of applying DOM changes to DOM tree may be in every animation frame, we are still keen on finding out the truth and potential changes in the benchmarking example. After researching and comparing several existing D3-focused virtual DOM frameworks [126][127], we choose react-

faux-dom [128] to be the test implementation as it provides a middle layer between D3 and virtual DOM (by React), namely D3.js is responsible for preparing the DOM in order to define the visualisation, and then React will be in charge of rendering the DOM for displaying.

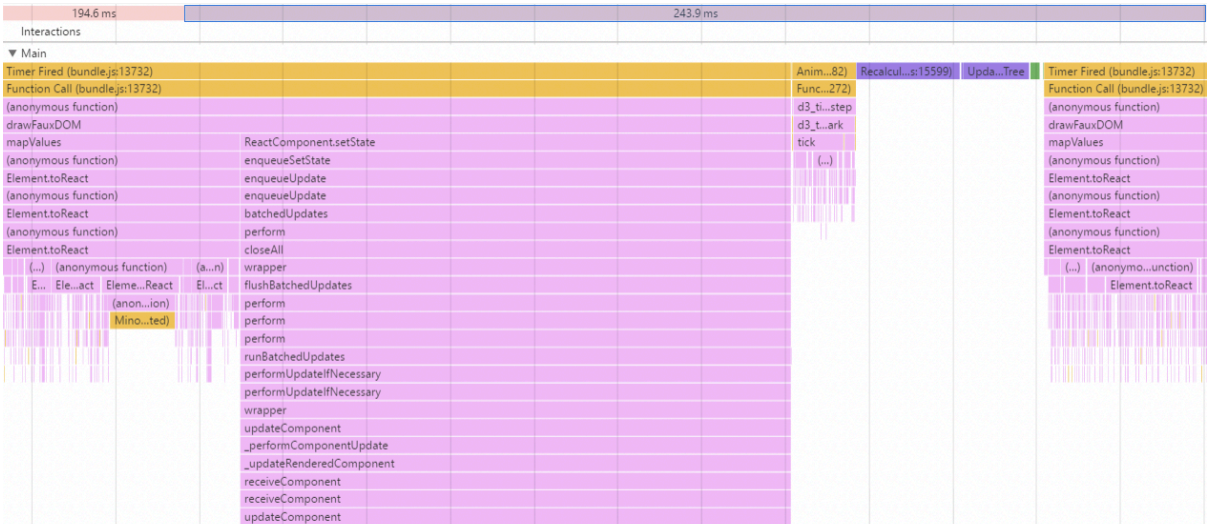


Figure 4.2.3 The screenshot of the timeline in benchmarking test using react-faux-dom

As we deploy the benchmarking test on D3 with react-faux-dom, we find that the animation frames become considerably long comparing to running on D3.js alone, where a single average frame of each is approximately 218ms versus 51ms in the test. In Figure 4.2.3, we can observe the problems are caused by frequently rebuilding the DOM tree in each animation frame [117]. Given the benchmarking tests were designed to animate randomly generated circles moving to random locations on the screen, it is reasonable to see that React.js completely updates the DOM structure in every frame in order to show the changes as necessary. To further address this approach, the possible alternative solutions will be discussed in the later section of §4.6.

4.2.3 Unoptimised Animation Timeout

Despite the fact that the React virtual DOM approach is not an ideal approach to optimise the animation-based visualisation such as in the last experiment, an interesting phenomenon was observed while experimenting and comparing the time intervals between animation frames in those two configurations. With React DOM, the idle time in its animation frames can be significantly shorter comparing to the ones in D3.js alone.

To take a closer look at the idle time differences, we find that D3 instead of using setTimeout [129] to trigger rendering the next frame of animation, it invokes requestAnimationFrame to decide the starting time for preparing the next frame. By looking into the documents of requestAnimationFrame function, it only updates the animation of the next repaint after receiving the callback from the browser, where this callback function is paused when running in the background or hidden, so that it can prevent the unnecessary animations to be played on the screen [130].

This default D3 timer configuration is reasonable for a personal computer environment (e.g. on a desktop or laptop), as the personal browser window may become inactive while a user switches the active main task to other browser windows or applications. Whereas, in the environment of Data Observatory, all of the visualising browser windows are constantly active and focused while presenting a visualisation application. Under these specific conditions, we are interested in to see the differences by setting a fixed time interval instead of checking the browser condition every time before deciding whether or not to render the next animation frame.

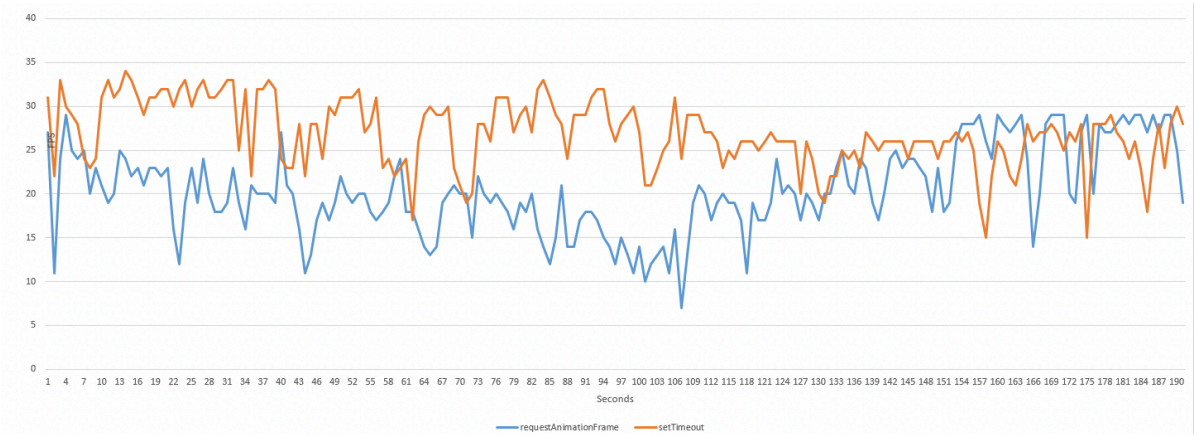


Figure 4.2.4 The comparison of frame rates (FPS) between requestAnimationFrame and setTimeout in the benchmarking test

In order to find out the possible performance impact while relying on one of these two configurations, we have run the benchmarking tests to show the potential difference in overall performance. As is shown in Figure 4.2.4, we can find the average frame rate (FPS) of using requestAnimationFrame in the benchmarking test is 20.4, which is to be compared with the frame rate of using setTimeout that is 27.4 [117]. The results demonstrate the possibility of performance improvement by 36.5% in the optimised settings. The underlying reason is potentially due to the fixed time interval set by setTimeout allows D3 to render the next frame of animation in advance via effectively reducing the overhead and waiting time of checking browser condition for each animation frame in requestAnimationFrame. The optimisation of D3 timer will therefore be implemented and detailed in the next section.

4.3 Distributed D3 Framework Optimisations

4.3.1 Optimising Animation Timeout

As we have discussed in the last section, the default D3 mechanism of the animation timeout in the current Distributed D3 framework may not be optimised. This is due to the conventional user environment has assumed to be on a personal desktop or laptop computer rather than in a visualisation-focused environment such as on the Data Observatory. The default D3 timer configuration can be easily changed by modifying the relevant code in D3.js as which is an open-sourced library. However, as we consider Distributed D3 to be a separate distributed layer based on D3, a preferred approach is to restrict the code modification within Distributed D3 if possible.

```
var d3_timer_queueHead,
    d3_timer_queueTail,
    d3_timer_interval,
    d3_timer_timeout,
    d3_timer_active,
    d3_timer_frame = this[d3_vendorSymbol(this, "requestAnimationFrame")] ||
function(callback) { setTimeout(callback, 17); };
```

Figure 4.3.1 The code snippet of initialising animation timeout function in D3 timer

By further looking into D3's source code when initialising the variables of its timer, we notice D3 has the backward compatibility to support the less updated browsers that do not recognise the requestAnimationFrame parameter, as we can see in the last line of Figure 4.3.1. The compatibility mode can be triggered if the requestAnimationFrame is *undefined* when initialising the D3 timer [117]. By checking the browsers and their versions on Data Observatory, we find this parameter are well supported in all browsers that have been used in benchmarking tests. Thus, a simple and effective workaround solution is to assign and force the requestAniamtionFrame parameter to be *undefined* before loading the D3 library in Distributed D3 in order to switch off this feature. The implementation results will be analysed and discussed in the section of §4.5.1.

4.3.2 Supporting D3 Version 4.0.0

As we have mentioned previously in the section of §4.2.1, upgrading Distributed D3 from the current version of 3.5.6 to the newer version of 4.0.0 could potentially resolve the performance issues on triggering excessive garbage collections, in addition to include possible new features in the current framework. In order to integrate the newer version of D3, we need to review and address a number of potential incompatible changes and issues in such an upgrade.

- *d3-ease methods simplified*: In the current version, d3.ease depends on strings for defining the ease of transition, which is used to define the transition speed changes during the transition, this is however changed in the newer version where the method names include the defining strings. For instance, a linear ease's method definition in the version of 3.5.6 and 4.0.0, is d3.ease("linear") and d3.easeLinear, respectively.

- *d3-collection methods renamed*: A number of methods have been renamed in d3.set and d3.map; for instance, the original method name of map.forEach has been changed to map.each for the simplicity. Such changes need to be checked and implemented in Distributed D3, as there are existing instances of method that may be not explicitly named after d3.set and d3.map.

- *d3-timer core changes*: Apart from the behavioural changes in D3 timer, which is now frozen in the background in order to avoid unintended effort, D3 is now able to use the high-precision time API performance.now() [131] rather than the previous API date.now(), which is also a key reason to support the newer version of D3 in Distributed D3.

- *d3-transition in a transition's life cycle*: It can be put that d3-transition is the main change in D3 v4.0.0 which has the most impact on the current development of the framework. There has been a large reworking of the transition object, and

the information now available for outside access is significantly less than previous. Besides, determining whether or not a transition is currently active becomes more difficult. In the next part, we will review the life cycle of a transition and also address the changes required for resolving possible issues.

The life cycle of a transition

Reviewing the life cycle of a transition in Distributed D3 can be useful for understanding the modifications that are necessary to be implemented in the framework to support newer version of D3. The following Figure 4.3.2 illustrates a transition's life cycle with the corresponding functions in steps.

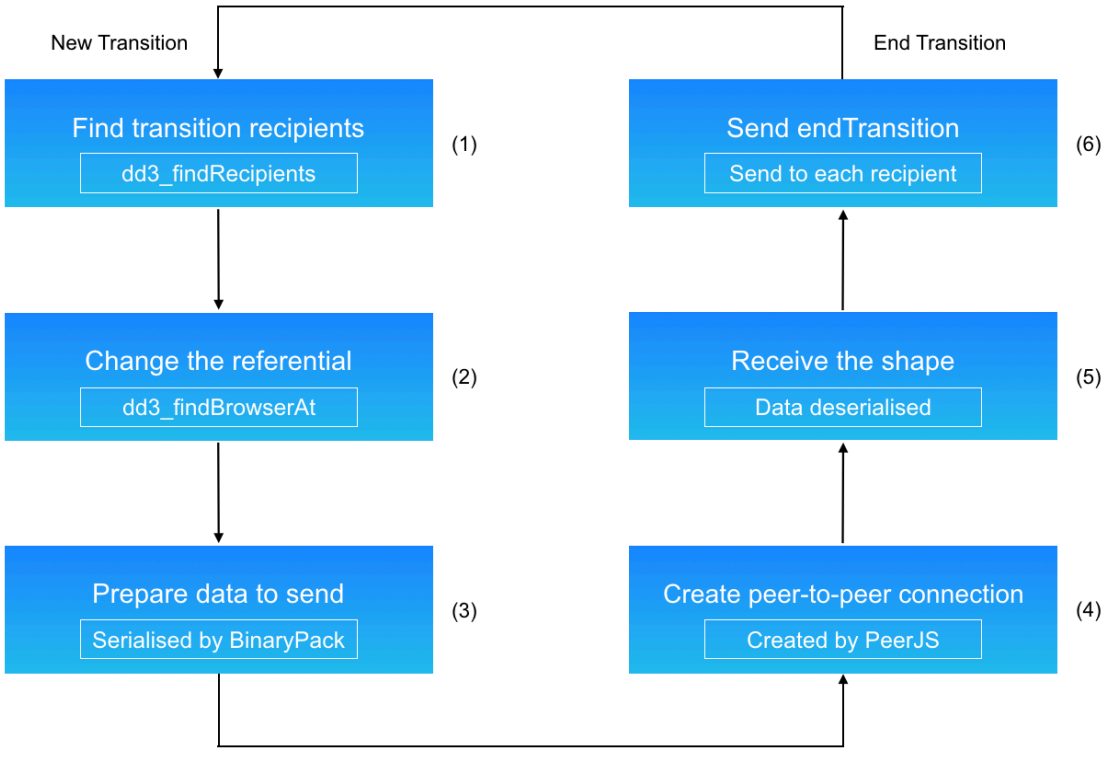


Figure 4.3.2 The life cycle of a transition in the current Distributed D3 framework

1. *Finding transition recipients*: As the transition starts, the framework will create elements which are supposed to be animated and prepare the shape to be transitioned. The dd3_findRecipients function will then utilise the native JavaScript getBoundingClientRect method to assist and find which browsers a shape will undergo during the transition.

2. *Changing the referential*: As the referential is present for every browser, it is important for the sender browser to change to a certain relative coordinate by implementing the dd3.position function, so that it can ensure the transition is visually understandable. The dd3_findBrowserAt function is responsible for this converting task.

3. *Preparing data to send*: As the BinaryPack [132] is used by PeerJS, which is the library utilised for peer-to-peer connections using WebRTC for serialising the data prior to sending it, as it is not possible to send unsupported elements including functions. Hence, it is necessary to serialise the transition-related functions such as tween(), attrTween() and styleTween(), as well as the ease() function.

4. *Creating peer-to-peer connections*: In case of the connection not being established with browsers, where the elements need to be received, the peer-to-peer connection is created by PeerJS. All data that is in need to be sent then fills the buffer once the connection is ready, and then the buffer is flushed and sent to other peers.

5. *Receiving the shape*: After receiving the transition, the recipient creates the element that the transition affects if it is not already present and interrupts the transition that exists. Moreover, it implements the style properties that are obtained from the peer and the initial attributes to this object. Then, it generates the transition object to which it applies the end style properties, the end attributes,

as well as the duration. Further, tween(), styleTween(), ease() and attrTween() functions are de-serialised and implemented to the transition object.

6. *Sending endTransition*: After the transition is completed, the sender which is the browser that begins the transition sends an endTransition event to each transition recipient, attributing the end state elements. It also functions as a mitigation mechanism if anything goes wrong during the transition on the other screen.

By reviewing the transition in Distributed D3, the first issue that needs to be addressed is how an active transition can be detected. In the current development, it depends on the transition object's *active* attribute. The newer version, however, has replaced this attribute by a global d3.active that returns the active transition but less reliable. Therefore, although the framework initially attempts using this method, if it is unable to detect any active transition, attempts will still be made to identify whether a non-null transition exists on the relevant object.

Secondly, the transition object's attributes are no longer accessible directly. Although the transition properties can be accessed using external code by implementing various workaround methods, a number of key attributes such as time, once the transition is created, remains inaccessible. This can be resolved, as these information are persistently stored in the node, which is to be transitioned and can be obtained by calling *transition*.node().__transition and retrieving all those transition properties.

Finally, *transition*.attr and *transition*.style help in establishing the transition's final state properties. While they do have their own behaviour previously, now *transition*.attr calls *transition*.attrTween function and *transition*.style calls *transition*.styleTween function as the adapted pre-defined interpolator functions. However, since Distribute D3 already includes the mechanism that can handle the transition without these updates, it is therefore necessary to remove them from the current development.

4.4 Distributed D3 Framework Demonstrating Application

4.4.1 The Design of the Demonstrating Application

In order to demonstrate the performance improvement and the usefulness of the Distributed D3 in a real-world use case scenario, we have designed and implemented a demonstrating visualisation application for the selected large dataset from a social speed dating experiment [133]. The dataset consists of 8,379 speed dates entries with at least common 75 attributes for each entry, which could provide a minimum of 627,750 elements in the demonstration application.

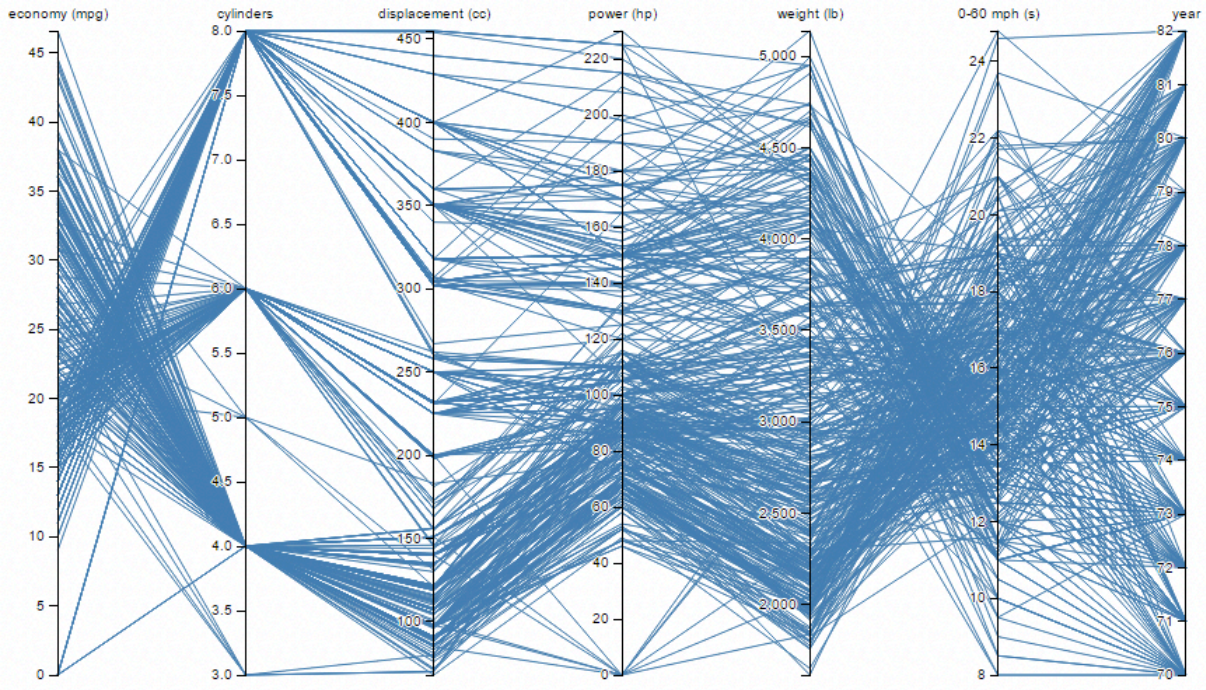


Figure 4.4.1 The visualisation example of the parallel coordinates with a dataset of cars

By considering the purpose in the design of this application, the demonstration should take advantages of using Distributed D3 in the environment of Data Observatory, i.e. the type of visualisation should be able to display raw data or information as rich as possible in the large tiled display wall comparing to the personal desktop environment. We found the parallel coordinates graph, which is a non-aggregated visualisation, can

be a good candidate that fulfils this purpose by allowing observers to directly observe and interact with the formatted raw data. An example of this type of visualisation can be seen in Figure 4.4.1 [134]. As a demonstration, we tend to keep the visualisation as simple as possible for the purpose of indicating the potential performance improvement by using a different number of raw data points from the dataset. The implementation of this demonstration will be discussed in the next section.

4.4.2 The Implementations of the Demonstrating Application

Since Distributed D3 preserves most of the D3 APIs for the simplicity of use, to implement this application on it becomes easy and effortless from the existing example. Essentially, we only need to modify the lines of code that are needed to use Distributed D3 and then adapt the modified example for the new dataset.

To be more specific, instead of using d3.scale, we use dd3.scale in both *x* and *y* scales. Further, in order to load data into the visualisation, we use dd3.getData to fetch data from database via the written OData services. After the callback with the result of successfully retrieving the data, we can draw and generate the visualisation as normally we do in D3.

Moreover, due to the uniqueness of the Data Observatory environment, using mouse to interact with the visualisation is inefficient. We thus have written a small control panel to enable the interactions that includes ranges and attributes filters when presenting data. We will present and discuss the implementation and benchmarking results of this application in section of §4.5.2.

4.5 Results

In this section, we will run a full benchmarking test for the optimised Distributed D3, and the test results will then be compared with the previous version to reveal the potential improvements. Meanwhile, we will also compare and discuss the potential changes in the visualisation capacity of the demonstrating application before and after the optimisations.

4.5.1 Benchmarking Comparisons

In order to compare the benchmarking results of Distributed D3 with and without optimisations, we have used the same benchmarking toolkit as we previously mentioned in §3.5.2. The benchmarking results have been shown in Figure 4.5.1, where we can find improvements in general comparing to the previous benchmarking results in the thumbnail with the same *x* and *y* scales at the bottom-left corner [117]. In particular, we can observe the noticeable difference in the configuration of the 16-screen setting, wherein the data line is completely flattened without declining in the optimised Distributed D3. This means the performance in this configuration at 2,000 animated elements is optimised and fixed at an ideal frame rate (FPS $\approx$ 60) during the test. Besides, we can also find the performance improvement in the 1-screen configuration, where the data line is now closer to the reference test result of running D3 alone (in red) than the unoptimised Distributed D3.

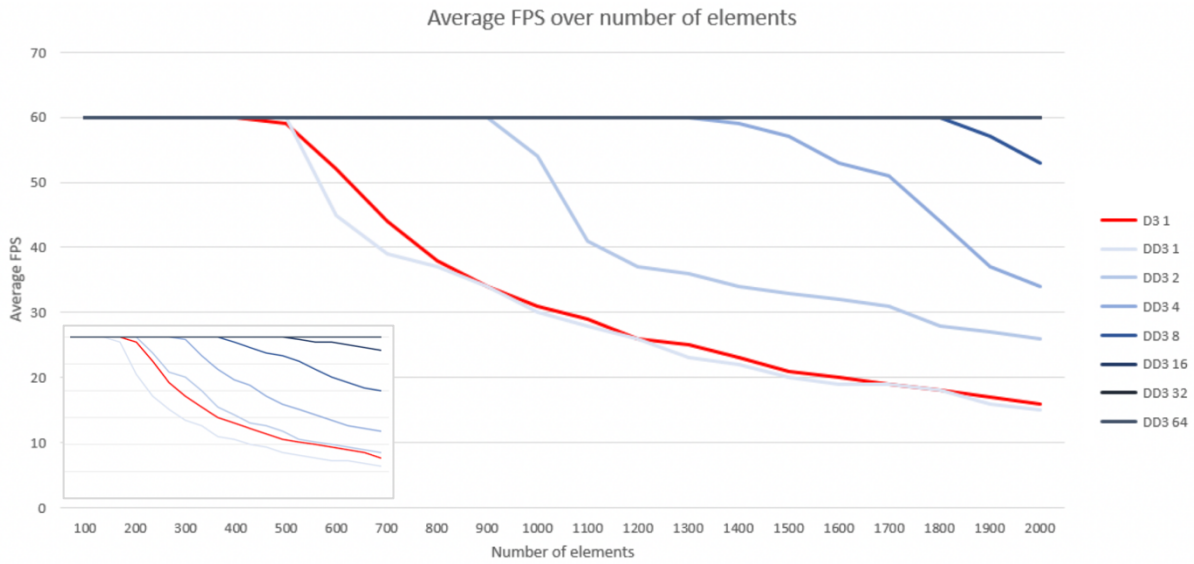


Figure 4.5.1 The benchmarking result of the average FPS for the optimised Distributed D3 in the configurations of 1 to 64 screens which is tested on the Data Observatory

To take a further look into the benchmarking results, we can separately compare the improvement details in the configurations of 2, 4 and 8 screens. As we can see in Figure 4.5.2 to Figure 4.5.4, the average frame rates (FPS) are noticeably increased by 15.9% in 2-screen, 22.8% in the 4-screen and 22.5% in the 8-screen configurations [117].

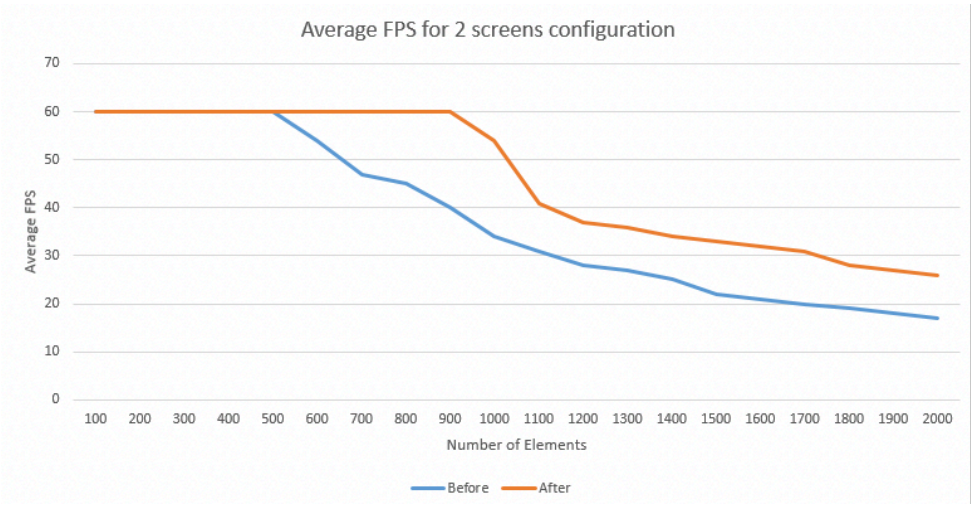


Figure 4.5.2 The benchmarking result of the optimised Distributed D3 in comparison with the previous version in the metric of average FPS for the 2-screen configuration

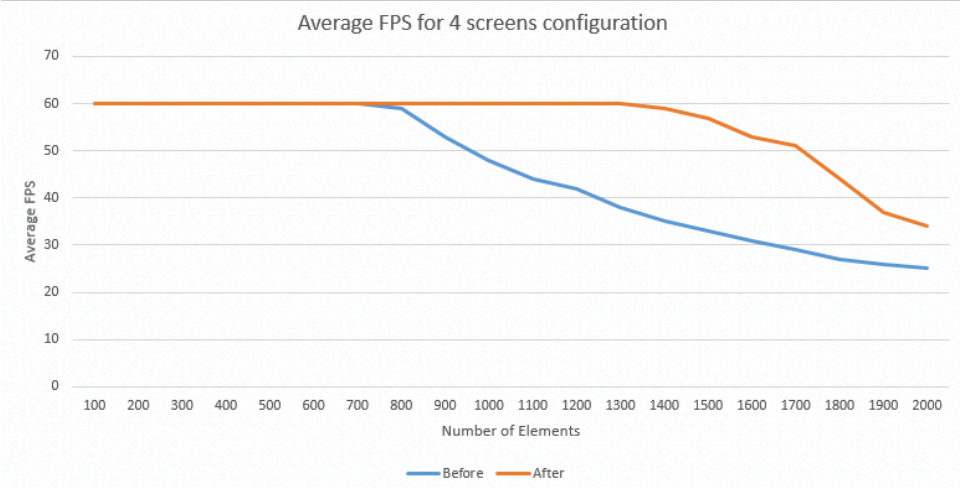


Figure 4.5.3 The benchmarking result of the optimised Distributed D3 in comparison with the previous version in the metric of average FPS for the 4-screen configuration

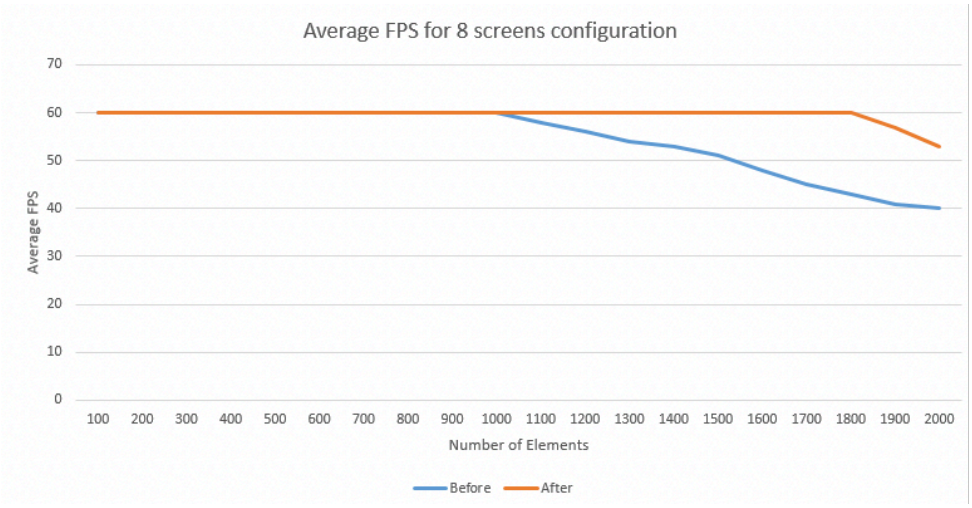


Figure 4.5.4 The benchmarking result of the optimised Distributed D3 in comparison with the previous version in the metric of average FPS for the 8-screen configuration

Moreover, we have also tested the performance limit of the optimised version in the configuration of 64 screens. The framework shows to be able to handle a maximum of 4,750 animated elements without performance decline in frame rate (FPS ≈ 60), which means an increased performance by 35.7% has been realised in the optimised Distributed D3.

4.5.2 Demonstrating Application

After we deploy the demonstrating application on the Data Observatory, we are able to observe 810 dates with a maximum of 38,070 elements at once on the previous unoptimised Distributed D3 framework. As we can see in Figure 4.5.5, the panoramic view of the data visualisation on Data Observatory greatly improves the visibility of the data trends and connections between different attributes in a larger visualisation area, which could help observers and collaborators to obtain the potential correlations more straightforward and easier [117].

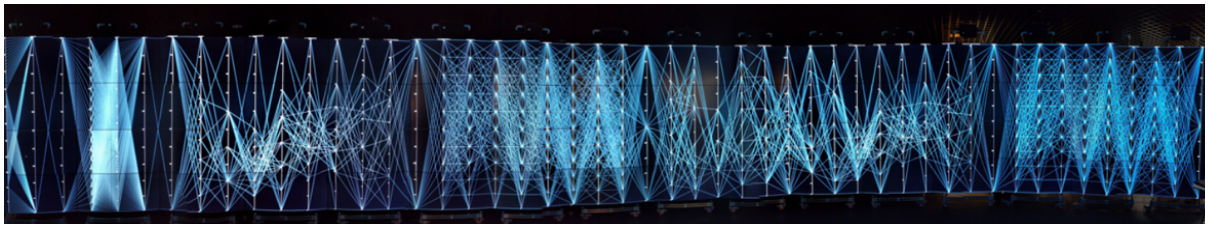


Figure 4.5.5 The visualisation result of deploying the demonstrating application on Data Observatory based on the unoptimised Distributed D3 framework with 38,070 elements

In comparison, when we deploy the demonstrating application on the optimised Distributed D3 framework, the Data Observatory with full 64 screens configuration is able to handle 1010 dates with 47,470 elements at once as is shown in Figure 4.5.6 [117]. The visualisation capacity has been improved by 24.7% in this demonstration, which also proves the usefulness of the framework optimisations in a real-world use case scenario.

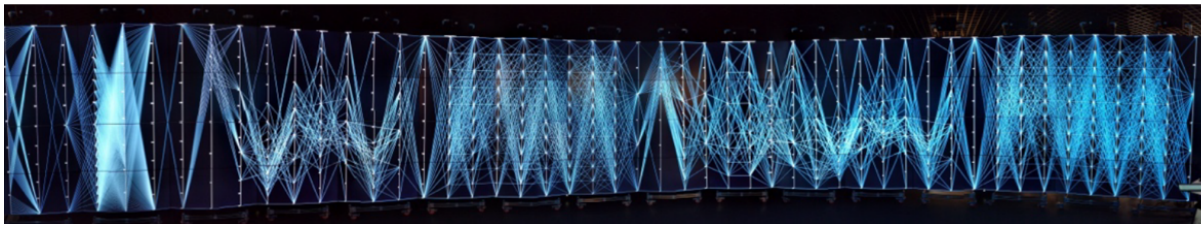


Figure 4.5.6 The visualisation result of deploying the demonstrating application on Data Observatory based on the optimised Distributed D3 framework with 47,470 elements

4.6 Discussion

As we have identified and experimented in the early sections, the overwhelming DOM access and interaction is a major bottleneck of both Distributed D3 and the original D3. If we can overcome the current implementing issue of completely rebuilding the virtual DOM to the DOM tree every time it deems necessary, we might be able to significantly reduce the performance impact from the excessive DOM interactions. Incremental DOM [135] is a recent development of the new emerging approach to address the high memory usage issue in virtual DOM. In the meantime, it would only incrementally apply the necessary changes to the DOM tree, which may help to resolve the issue of repeatedly rebuilding DOM tree in the previous experiment. If this future work can be realised with a good benchmarking result, it can be expected to be a remarkable breakthrough on the performance bottleneck issue in both Distributed D3 and D3.

The animation timeout mechanism is optimised by switching the function from the default requestAnimaitonFrame to setTimeout based on the operating characteristic of the Data Observatory. The implementing results of this optimisation are remarkable. However, we do realise the potential animation issue when multiple instances of visualisation are running on the Data Observatory simultaneously. In which case, requestAnimaitonFrame may handle this situation in a more reliable and managed manner due to its scheduling mechanism [136]. Meanwhile, the reliability could also suffer if the animation timeout interval is not fit for the number of elements in a visualisation. To address these concerns, we can build a customised timeout module that smartly switches and adjusts these functions and parameters based on the current number of visualisation instances and the number of elements in a visualisation.

4.7 Conclusion

In this chapter, we have investigated the performance bottleneck of the framework, where two bottlenecks were identified from the performance analysis. We have addressed the first bottleneck of excessive garbage collections by integrating the newer version of D3 4.0.0 into the Distributed D3. We have also tried to address the second bottleneck of massive DOM interactions by implementing react-faux-dom into the framework, wherein the test results are less satisfying; however, we noticed a less optimised animation timeout mechanism that is caused by the function of requestAnimationFrame. The final benchmarking results confirm and show the performance improvements of the optimised framework by 35.7%.

Apart from optimising the framework, we have further designed and implemented a demonstrating application for a real-world use case scenario. In which case, we selected a random large dataset from a social experiment and then built am illustrative visualisation based on that. We use this application not only to demonstrate the usefulness of Distributed D3 in reality, but also to show the improvements of the optimised framework comparing with the previous version. The future improvements and developments of the framework are also discussed, where we proposed the new scheduling mechanism for the animation timeout and the potential solution of the incremental DOM to address the issue of massive DOM interactions, particularly in the large-scale data visualisations.

Chapter 5 Distributed D3 Framework Generic and Standalone Versions

5.1 Introduction

Distributed D3 is a novelty distributed data visualisation framework that provides unique features of utilising the cluster computing power to improve the performance and scalability of the original D3 library. With the recent developments and optimisations of the Distributed D3 framework, we have delivered the first integrated version that can work together with Data Observatory (DO) to realise the distributed visualisation, and later the optimisations and upgrades to the integrated version that further improves the framework performance not only revealed in the benchmarking tests, but also demonstrated in a large-scale data visualisation application.

However, the current developments of Distributed D3 is highly integrated and dependent on DO, which could also limit the usability of the framework on other visualisation environments. In order to make the framework independent, we realise currently the primary attachment to DO as an application is the implementation of the SignalR hub network by default. The historical reason was the operating framework of DO is written in C#, which can be well supported by deploying a SignalR server on it. Therefore, we plan to replace the SignalR hub with a generic independent server for the detachment, and then take the step further to completely remove the server by implementing the serverless network design for the additional flexibility in generic version.

In this chapter, we will detail the detachment and serverless design and implementations for the generic and standalone versions of the framework. The implementation results

will be demonstrated by configuring the framework on a customised environment. We will also conduct the benchmarking tests for the generic and standalone versions on DO and the customised environment for comparisons. At the end of this chapter, we will discuss the potential and known issues in the current development before conclusion.

5.2 Generic Distributed D3 Framework Design

In order to achieve the generic and standalone design of the Distributed D3 framework, we have designed the two-step approach that aims to improve both usability and flexibility of the framework. The first step is to detach the Distributed D3 framework from the integration of Data Observatory by replacing the SignalR hub with an independent server, and the second step is then to remove the server completely by assigning a master node instead. We will detail these two steps in the following subsections of §5.2.1 and §5.2.2.

5.2.1 The Detachment Design

In the current design of Data Observatory (DO), all of its applications are instructed to include the file structure of App and AppHub in order to fulfil the communication needs between distributed nodes and the central server. Since the main operating framework of DO is written in C#, SignalR [137] is thus well supported in such an environment (as SignalR hub in AppHub). In facts, most of the existing applications on DO are deployed a SignalR server for various reasons. Hence, the main objective in the detachment becomes to find a suitable independent server that is able to provide the same features as the SignalR hub can do. Meanwhile, the relevant application interfaces need to be created and maintained while replacing with a new server.

If we look into the existing application interfaces between the SignalR hub and the distributed nodes, we can find their structure and connection as illustrated in Figure 5.2.1, where the SignalR hub (server) is mainly responsible for broadcasting the packed information to nodes, including the individual node configurations for the peer network and the controlling commands from a controller (i.e. control node) [117]. Since we also have plan to adapt a serverless pure peer-to-peer network in the later development, a good approach is to manage the application interfaces by designing and creating an

extra network interface layer that can flexibly handle the network switching. Such a network interface will be detailed in the section of §5.3.1.

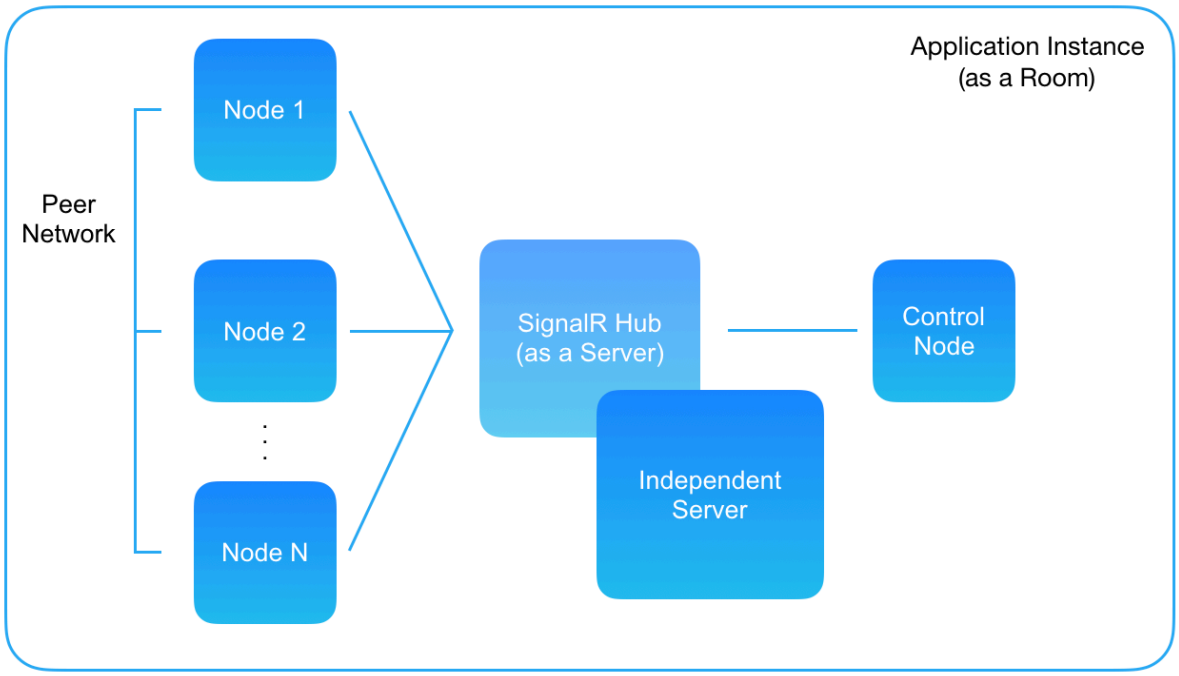


Figure 5.2.1 The architecture of replacing the SignalR hub with an independent server in order to detach the framework from the integration of Data Observatory

In addition, since the SignalR server is able to handle the multiple application instances by dividing the connected nodes into groups as its integral feature, the same mechanism needs to be addressed in the independent server design. The room concept can be introduced to deal with this issue for each application (i.e. one room for one application). Hence a node will create or join a room while connecting to the independent server, and this node will be removed from the room if it is disconnected from the server. In which case, the multiple application instances can be handled simultaneously by having their own separate rooms. The implementation of this design will be detailed in the section of §5.3.2.

5.2.2 The Serverless Design

While there are several advantages of designing the underlying network with a central communication server in terms of dedication and reliability; the serverless and pure peer-to-peer network design may also have its unique advantages in the aspects of framework simplicity and scalability, where the framework is able to scale flexibly without limitations from a server. In the meantime, the unnecessary components that are related to the server can be removed to promote a lighter weight framework. The design of the first serverless network structure will be illustrated and detailed in this section.

The main design philosophy behind this initial serverless version is to completely remove the server with the support by a master node as shown in Figure 5.2.2 [117]. This master node plays an essential role that acts as a server to broadcast the relevant node information and controlling commands, which is also known as a super-peer in the literature [64]. Indeed, the super-peer design may potentially increase the workload on that assigned rendering node as a master; the overall performance impact, however, needs to be further evaluated and compared with server dependent design in order to draw a sensible conclusion. Meanwhile, as the first serverless release of the framework, further improvements will be in need to reduce the potential performance impact on the master node, as well as to enhance the reliability of the network in case of the master failure. The implementation detail of this first serverless design will be discussed in the section of §5.3.3.

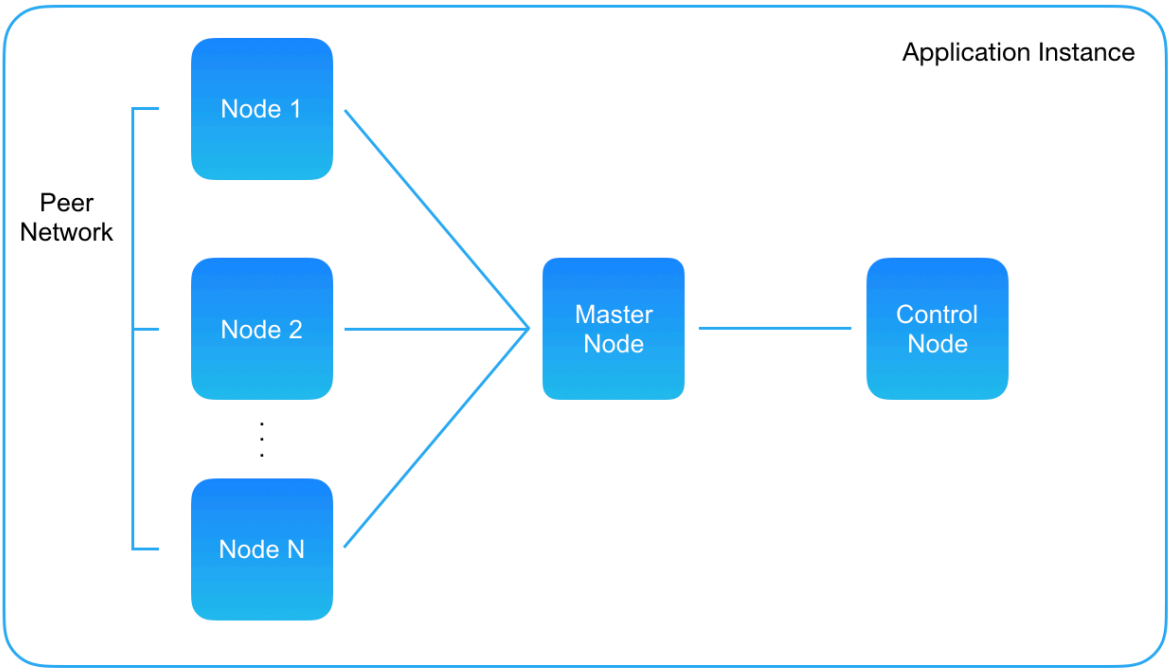


Figure 5.2.2 The architecture of the serverless design by assigning a master node and removing the independent server from the framework

5.3 Generic Distributed D3 Framework Implementations

5.3.1 The Network Interface

In the design of the generic and standalone Distributed D3 framework, we proposed two steps for detaching and decentralising the framework from the integration. Such modifications of the existing framework can be benefited from defining a common network interface for diversions of the framework. We thus redesign the current network APIs to include an extra abstraction layer of the network interface, for the purpose of switching networks based on requirements as we can see in Figure 5.3.1, where the current design of the network interface allows to switch between SignalR-based, SocketIO-based and pure PeerJS networks [117]. Such a design further enables the possibility of mixing hybrid networks and improves the flexibility of utilising the framework in different usage scenario.

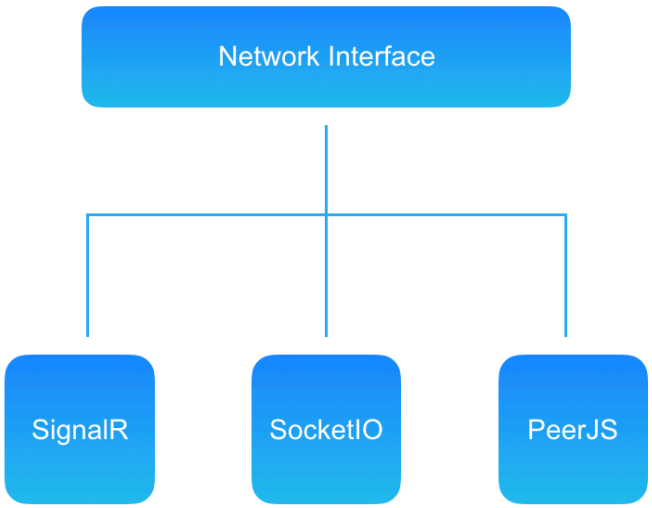


Figure 5.3.1 The abstraction layer of the network interface for the purpose of switching networks on demand, which also allows to hybridise the existing networks if needed

Corresponding to the network interface design above, we have designed the superclass of NetInterface to include the common properties and methods that are shared by diverse network protocols, and each network protocol has its own subclass that additionally includes private properties and methods for that particular protocol in use. For example, in the subclass of PeerNet, it has the peerOptions that contains the peer naming server and port for registering and obtaining peerIds.

The realisation of this abstraction layer at code level can be seen in Figure 5.3.2, where we employ the JavaScript prototype to build the network interface classes. Hence a variety of network protocols can be initialised as the network interface instances by using a common class constructor. At this stage of development, we have included the instances of pure PeerJS, SignalR-based, SocketIO-based networks. Meanwhile, in order to enable the inheritance in JavaScript with prototype, we have used an *extend* function to inherit the class properties and methods from a superclass to a subclass [138]. In which case, the subclass of PeerNet is extended from the superclass NetInterface as we can find in the second part of Figure 5.3.2 [117].

NetInterface:

```
function Network(protocol, config) {
    var protocol = protocol || null;

    if (protocol === "peerjs")         // pure peer network
        return new PeerNet(config);

    else if (protocol === "signalr")   // signalR-based net
        return new SignalrNet(config);

    else if (protocol === "socketio") // socketIO-based net
        return new SocketioNet(config);

    else
        throw new Error('Protocol not existed');
}
```

```
function extend(subClass, superClass) {  // extend function
    function F() {};
    F.prototype = superClass.prototype;
    subClass.prototype = new F();
    subClass.prototype.constructor = subClass.constructor;
}

function NetInterface(config) {        // class constructor
    this.config = config || {};
    this.browser = {};
    this.utils = {};
}

NetInterface.prototype.setBrowser = function (browser) {}
NetInterface.prototype.setUtils = function (utils) {}
```

PeerNet:

```
function PeerNet(config) {
    NetInterface.call(this, config);  // inherit from superclass

    this.id = this.config.id || '';   // private properties
    this.peers = [];
    this.connections = [];
    this.peerOptions = {};
}

extend(PeerNet, NetInterface);  // extend subclass PeerNet
                                // from superclass NetInterface
```

Figure 5.3.2 The pseudocode of network interface class with an example of the pure peer network protocol subclass

5.3.2 The SocketIO Approach

In order to replace the SignalR server, we have found the independent NodeJS-based real-time web application framework – SocketIO [139] can be a good candidate in this implementation. It features by its flexibility and scalability of building a portable server, while providing fast and reliable real-time service [140].

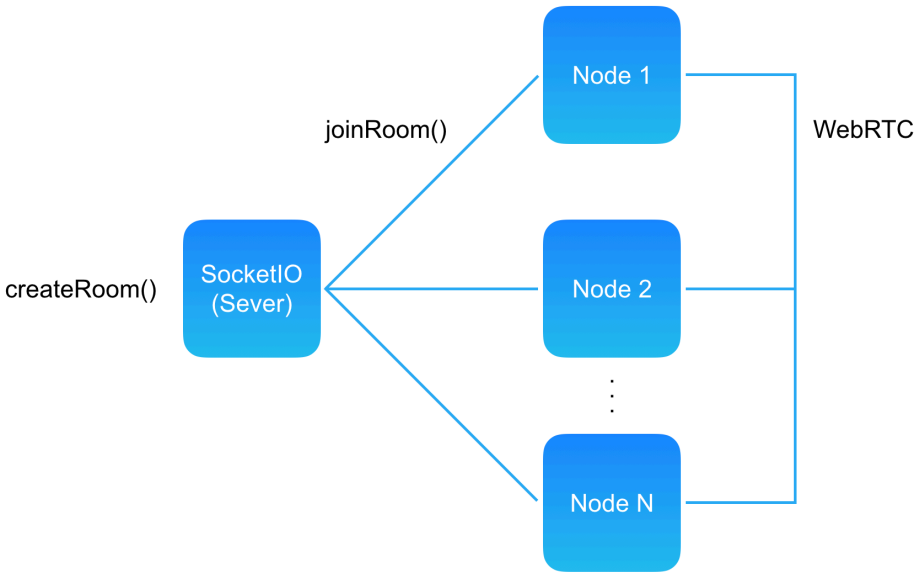


Figure 5.3.3 The illustration of replacing the SignalR hub (server) by a SocketIO server

The original SignalR hub is replaced by a SocketIO server in this implementation as shown in Figure 5.3.3 [117]. We implement the room concept for the purpose of enabling the multiple visualisation instance scenario, where a room is uniquely identified by the roomId that is in the combination of applicationId and the unique instanceId. The individual node's socket can be stored and identified by the socketId when the connection is established, and it will be removed when the connection is lost. The socket can be further used to join and leave a room for the group broadcasting as we can see in Figure 5.3.4 [117].

Specifically, once the SocketIO server is established, it creates a room that is listening and waiting for the nodes to be joined. When a node is initialising, instead of connecting itself to the SignalR server, it attempts to join the SocketIO room via the newly designed network interface. Once all nodes are joined, the server will then broadcast and update the node information, which is the same as the SignalR server would normally do. After that, the fully connected peer-to-peer network can be established once the node's information is received by each peer node.

SocketIO:

```
var members;                              // global scope

io.on('connection',function(socket){

  members[socket.id] = socket;            // store new member

  socket.on('disconnect', function(){

    lib.removeMember(socket);             // remove a member
                                          // if disconnected
  });

  socket.on('joinroom',function(data){

    var roomId = '' + data.applicationId + data.instanceId;
    // define the roomId by appId and instanceId

    lib.joinRoom(socket, roomId);         // join the room

  });
})

var lib = {

    joinRoom(socket, roomId){
      socket.join(roomId);                // join by roomId
    },
```

```
    removeMember(socket){
      socket.leave(socket.roomId);     // leave by roomId
      delete members[socket.id];       // delete the member
    }
}
```

Figure 5.3.4 The pseudocode of creating, joining and removing from a room in SocketIO

5.3.3 The Pure PeerJS Approach

In comparison with the SocketIO approach, the implementation of the serverless pure peer network design requires to define a master node as a representative to broadcast the timely information for other nodes. We have discussed the network structure and the advantages of implementing an additional star network comparing to use a single fully-connected peer network in the section of §5.2.2. Since we have used the PeerJS to build the peer network among rendering nodes in the previous developments, and we had positive experience and feedback of implementing this WebRTC-based library on the framework in terms of speed of connection and network reliability, we therefore decide to implement the pure peer network structure entirely with PeerJS, which is also for the overall consistency and simplicity in the aspects of the main framework components.

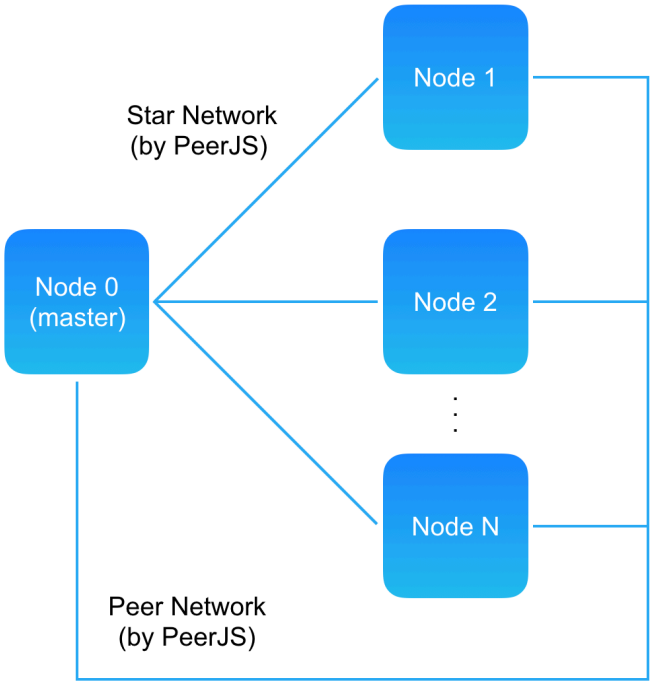


Figure 5.3.5 The illustration of implementing the serverless pure peer network by PeerJS

The structure of the serverless pure peer network is illustrated in Figure 5.3.5, which is implemented by using the PeerJS framework [117]. In which, the first node (Node 0) is assigned to be the master node by default, who will be in charge of establishing the star network at the initialising stage. After all peer nodes are connected, the master node then broadcasts the node information array in order to help establishing the fully connected peer network afterwards. The implementation detail of the network will be discussed below.

In detail, the client-server network structure allows the node configuration information to be easily collected and maintained while a new node is connecting to the server via a given server address. This step however is lacking in the pure peer-to-peer network structure, as all peerIds are dynamically generated (a fix and static peerId is possible, but it is less elegant and not feasible in the case of running multiple application instances). In order to address this issue, we have found the peer discovery function listAllPeers in PeerJS, which can be used to fetch the list of registered peerIds and is particularly helpful for the master node to establish the star network in the initialising phase. To be more specific, when new peers are connecting to the PeerJS naming server to obtain their peerIds, their existence are registered, hence a master node can fetch the list of registered peers that are identified by their individual peerIds.

To further simplify the steps in the network initialisation, we have designed the structured peerId to consist of the node configuration information. For instance, a designed peerId with comprised fields would be expressed as,

```
ctrl3_conf0_client_r3_c0_node1_1571762550369
```

where the separated fields are controlId, configId, client (or control), the node's row number, the node's column number, nodeId and the timestamp when this peerId created, respectively.

A unique application instance can be identified by the combination of the controlId and configId. Such a structured design in every peerId not only reduces the possible

initialising steps via network communications, but also allows to quickly identify a group of peers in the case of running multiple visualisation instances in one visualisation environment.

As the JavaScript pseudocode outlined in Figure 5.3.6, in order to establish the star peer network, the master keeps pooling the connected peers every second (1000ms) at the initialising phase [117]. The fetched list of peerIds with node information is extracted and prepared for broadcasting in the master node. In the case that the given clientNum is equal to the number of connected peers in the list, the star network is established by the master node; meanwhile, the node information array is also broadcasted. Finally, the individual peer node information in the array are used to establish the fully connected peer network afterwards, in which case, each peer node iterates and connects to all peers in the array except itself.

```
Master Node:

var infoAry = [ nodeInfo ];  // initialise info array with
                             // storing master node info

poolingConnectedPeers(){
  var peers = peer.listAllPeers(function(list){
    list.forEach(function(l, i){
      ls = l.split("_")       // extract node info
      infoAry.push(ls);       // store info into array
    }
  });
}

initStarNet(){
   if(nodeId = 0){                    // if a master node
     if(clientNum = infoAry.length){  // if all connected
       foreach(info in infoAry){
         connectToThisPeer(info.peerId);
         // connect to all other peers by master
       }
```

```
        initPeerNet();                  // for master
      } else {
        poolingConnectedPeers();        // every 1000ms
      }
    }
}

initPeerNet(){
  foreach(info in infoAry){
    connectToThisPeer(info.peerId);
    // connect to all other peers for master
  }
  waitingToBeConnected(function(){
    // waiting to be connected by other peers
  }
}
```

Peer Node:

```
var infoAry;

initStarNet(){
  waitingToBeConnected(function(data){// await master
    infoAry = data;                 // store info array
    initPeerNet();                  // initialise peer net
  }
}

initPeerNet(){
  foreach(info in infoAry){
    connectToThisPeer(info.peerId);
    // connect to all other peers by each node
  }
  waitingToBeConnected(function(){
    // waiting to be connected by other peers
  }
}
```

Figure 5.3.6 The pseudocode of initialising the pure peer network by assigning a master node

5.4 Generic Distributed D3 Framework Demonstrations

5.4.1 The Configurations on a Customised Environment

Following the development of the generic version of Distributed D3, a test experiment is designed to deploy the independent framework on a customised visualisation environment, which consists of a small cluster of 3 desktop computers with 3 screens in a row. The hardware specifications of each computer are, Intel i5-7200U (2.5 GHz) with 4Gb RAM, 2Gb Intel HD graphics card with HD screen (1920x1080) and operating on 64bit Windows 7.

Specifically, to set up Distributed D3 on this customised environment is simple and fast. First of all, we need to set up the NodeJS [141] environment for the supporting frameworks including PeerJS, SocketIO and Express [142], and then MongoDB is required as the main database in this version, finally we can download the framework by checking out the source code from the repository.

1. Install NodeJS, in order to include the supporting frameworks like PeerJS, SocketIO and Express, where Express is used to host the framework locally.

2. Install MongoDB, which is supported by the developed OData service modules in Distributed D3, and data files also need to be imported to the database.

3. Deploy Distributed D3, by checking out the source code from the repository, and then the framework is ready to use.

At this point, we can simply run commands to launch the framework for the journey of visualisations.

5.4.2 The Demonstrations of the Customised Environment

For the purpose of testing the configuration result of this environment, we deploy the fundamental charts examples that are used previously in section of §3.5.1.

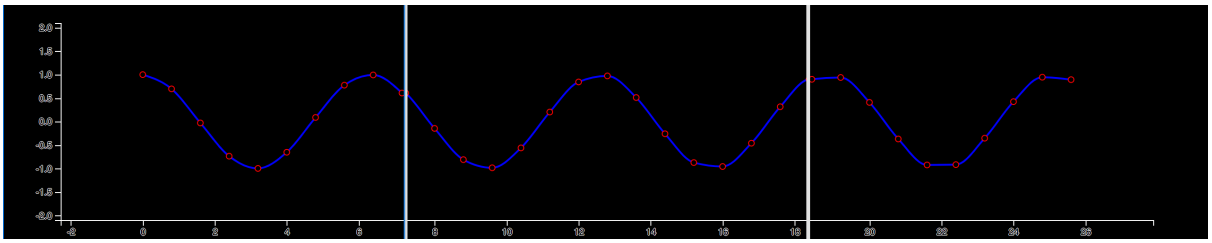


Figure 5.4.1 The screenshots of demonstrating the scatter plot example on the small customised environment with 3-screen setting

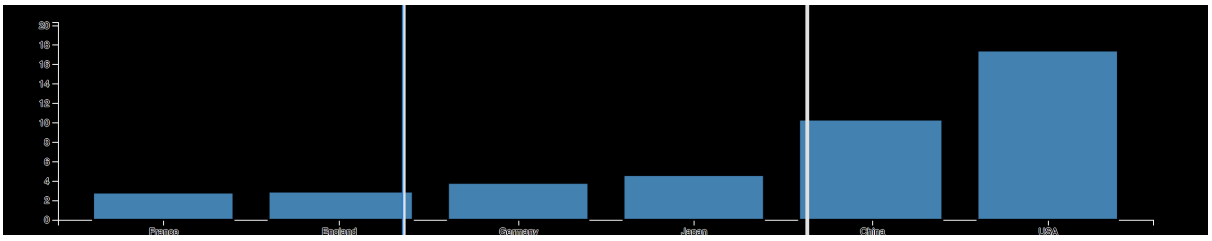


Figure 5.4.2 The screenshots of demonstrating the bar chart example on the small customised environment with 3-screen setting

As we can see in Figure 5.4.1 and Figure 5.4.2, the demonstrating examples show that the generic version of the pure peer network Distributed D3 has been successfully configured at its working condition on this customised environment [117]. We will run a further benchmarking test as a comparison to the Data Observatory for this environment in the section of §5.5.2.

5.5 Results

5.5.1 Benchmarking on Data Observatory

For the purpose of revealing the potential performance changes in the generic versions, we have made comparisons between two standalone versions. To keep the metrics consistent, we have used previously developed the FPS animation benchmarking toolkit for the tests, which has been discussed in §3.5.2. The test results can be found in Figure 5.5.1 and Figure 5.5.2.

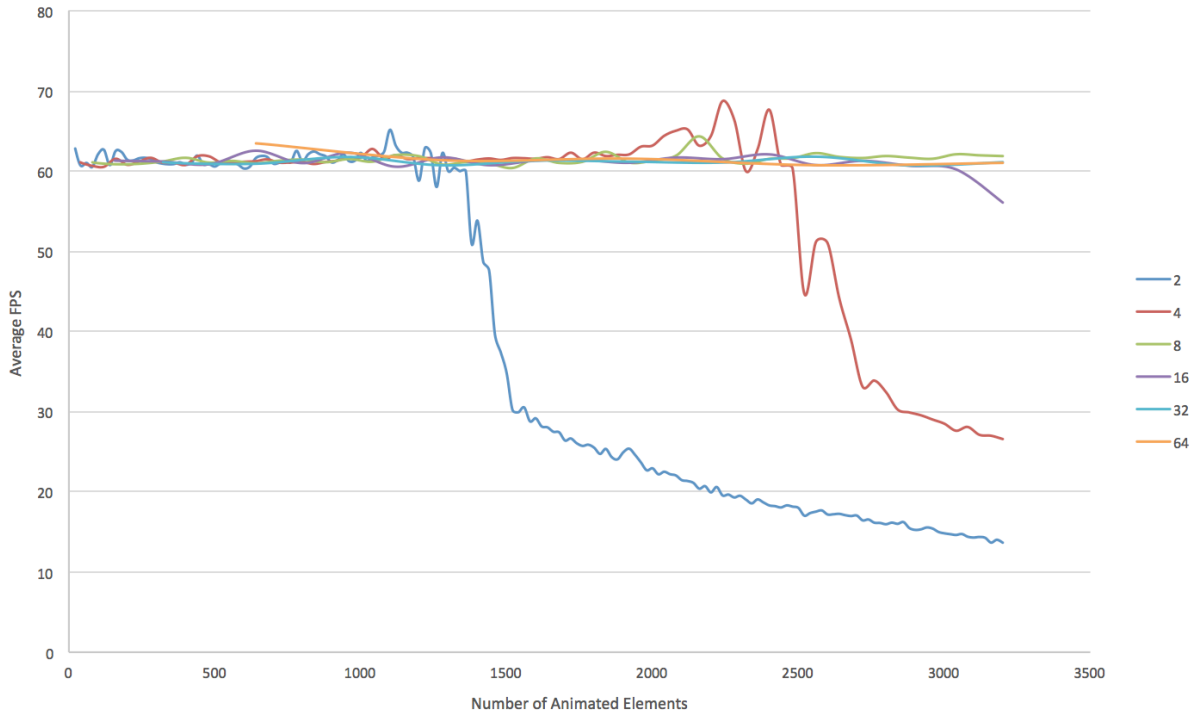


Figure 5.5.1 The benchrmarking result of the generic Distributed D3 with the independent server network implemented by SocketIO

Figure 5.5.2 The benchrmarking result of the generic Distributed D3 with the pure peer-to-peer network implemented by PeerJS

By comparing the Figure 5.5.1 and Figure 5.5.2 with the previous benchmarking result in Figure 4.5.1, we can find that the standalone versions are significantly outperforming the integrated version on all the configurations with a different number of screen settings [117]. The pure peer network (PeerJS) version is also slightly outperforming the client-server (SocketIO) version, which can be observed on the configuration of the 8-screen setting. The potential performance difference is likely caused by the framework operating overhead in the integrated version. Meanwhile, the pure peer network version is also lighter than the client-server version, in terms of the serverless design.

5.5.2 Benchmarking on Customised Environment

Apart from benchmarking the generic framework on Data Observatoy, we have also run the benchmarking test on the customised environment in order to find the potential differences and issues from the test. Since the serverless pure peer-to-peer implementation is the preferred approach in the design of Distributed D3, we mainly focus on evaluating this implementation in the following test.

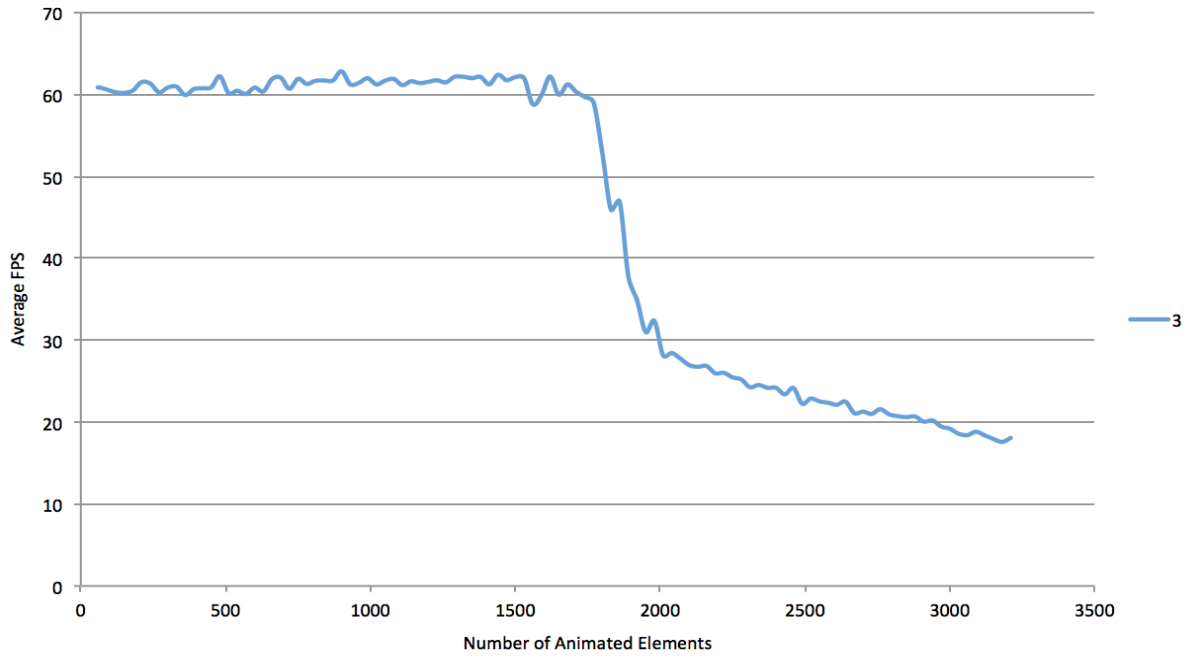


Figure 5.5.3 The benchmaking result of the generic Distributed D3 (with pure peer network) which is tested on the cutomised visualisation envinronment with 3-screen setting

In Figure 5.5.3, we can observe a similar trend as the benchmarking result of the pure peer-to-peer network version on the Data Observatory [117]. We notice the declining point of the data line is around 1,700 animated elements, which is only slightly outperforming the 2-screen configuration on Data Observatory in Figure 5.5.2. The results are reasonable as there are 3 screens (i.e. 3 browser windows) in the customised environment; meanwhile, the individual graphical stations on Data Observatory are more powerful than the desktop computers in this customised environment.

5.6 Discussion

From the benchmarking results of the independent client-server and pure peer-to-peer network structures, we can see the advantages of using the peer-to-peer network to construct the framework, in terms of overall performance and simplicity in components. However, we are aware of the drawbacks of implementing the pure peer network with the super-peer design, where a master node (i.e. super-peer) is assigned to take the responsibility of a server.

The super-peer can potential become the single point of failure for the whole cluster and the potential bottleneck, in which case if a super-peer fails, it can lead to the disconnections of all other rendering nodes until they find a new super-peer to be connected with. A possible solution is to implement the super-peer redundancy to improve the reliability, whereas it may also come at a cost in the network performance [64]. Therefore, we believe further research in this area might be needed in order to find more favourable solutions.

In addition, as we know D3.js is based on a very active open-sourced community with a large number of contributors and collaborators, the iteration of developing new features and thus new versions of D3 is very fast. Since Distributed D3 is a distributed framework that is based on D3, we may need to take account of the changes and updates in order to support the latest stable D3 for the benefits of users. We are in hope to address this development issue by publishing and creating the open-source community of Distributed D3 to further update and improve the framework with more collaborators.

5.7 Conclusion

In this chapter, we have proposed and designed the standalone versions of the Distributed D3 framework. Two branched versions are implemented by SocketIO and pure PeerJS. The benchmarking results show that both versions are faster and light-weighted compared with the integrated version. The pure PeerJS version is also slightly faster than the SocketIO due to the light-weight real-time peer-to-peer data channels based on WebRTC.

The test experiment of setting up the Distributed D3 standalone version on a customised small cluster of desktop computers confirms the framework can be flexibly configured and utilised on a variety of visualisation environment depending on the requirements. Such a customisable feature of the framework may especially benefit the data visualisation community with an existing scalable high-resolution display environment.

The development of the standalone version of the Distributed D3 makes the framework more accessible and useful for the open-source community. The release of the standalone version may further improve the usability of the framework. We hope this could fill the gap between the data visualisation and distributed system in the web environment, and hence to advance the visualisation technologies in this research field.

Chapter 6 Conclusions

The rapid growth of data in Big Data era has increases the need for analytic tools to obtain insights from large datasets. Such data sources may range from Internet of Things sensor networks to the growing Open Data movement. Visual perception being humans' primary ability offers the distinct ability of rapidly differentiating patterns in a pre-attentive manner. Hence, data visualisation for visual analytics is a powerful tool that has become a significant discipline. Today, D3.js has become a powerful web-based data visualisation library that can be the standard tool to visualise data. The library, however, is technically inherent, limited in its ability, as well as being unable to deal with large datasets.

This thesis focused on overcoming this limitation and resolving the challenges through the development of the Distributed D3, which uses the distributed mechanism that can help generate web-based visualisations for large datasets based on D3.js that also helps in effectively using the graphical computational resources of the modern visualisation environments. The work mainly intended to provide a robust Distributed D3 that can preserve the API compatibility of D3.js library for its simplicity of use. Therefore, the framework resolved D3.js's performance bottleneck that hindered the visualisation of large-scale data, thereby enhancing D3.js's overall scalability as well as usability. The framework also helped in diverse visualisation environments being configured and programmed by an extensive community of developers for collaboration and research.

The specified main contributions in the development of Distributed D3 are:

- In Chapter 3, we present the integrated version of Distributed D3 framework for the Data Observatory. We compare the different designing approaches in order to properly design the framework for implementations. The implementation results show that the framework overcomes the performance limits of D3.js,

which also proves the concept of Distributed D3 that is feasible. Meanwhile, the framework is also benchmarked to evaluate the improvement of the overall performance and scalability compared with the original D3 with demonstrating examples. This work further enables a wide community of developers to be able to collaborate on large-scale data visualisations in the Data Observatory.

- In Chapter 4, we present the optimised and upgraded version of Distributed D3 framework for large-scale data visualisations applications. We investigate the potential bottlenecks of the existing Distributed D3 in order to optimise the framework and address the underlying issues. The optimisation benchmarking results show that an improvement of the overall performance by 35.7% is achieved and compared with the unoptimised Distributed D3. Meanwhile, the support of newer D3 also extends the functionality of the framework for potential applications. A demonstrating application is presented for the purpose of illustrating these improvements in this version. This work therefore further improves the scalability and usability of Distributed D3 for the visualisation applications with large-scale data.

- In Chapter 5, we present a generic version of Distributed D3 framework for the customised environments on demand. We propose the detachment and serverless design of the framework in order to enable it to be fully independent for generic uses. The implementations improve the flexibility of the framework by allowing switch networks between classic server-based and pure peer-to-peer when necessary. The benchmarking results show that the version is light-weighted and slightly faster than the previous versions. A customised visualisation environment is also set up for the demonstrating and comparing purposes for this version. The work improves the usability and flexibility of the framework and makes it ready to be published in the open-source community for further improvements and usages.

With the uniqueness of the framework design, it can provide a novel solution to the modern data visualisations especially for large datasets. It is therefore in hope to be able to contribute to the open-source community by advancing the field of large-scale data visualisation with distributed approach.

Future Works

The academic papers of Distributed D3 framework are in plan for detailing and demonstrating the approach including the structures and algorithms that are designed to enable the main distributed features of the framework. Meanwhile, we have the plan to publish the framework source code and create an online open-source community of Distributed D3, which is building upon the existing and very active D3.js community.

Apart from the relevant works in publications, the framework can be further improved by addressing the remaining issues and challenges in the future work,

- In large-scale data visualisations, the overwhelming DOM access and interactions are the main bottlenecks of D3.js and thus Distributed D3. Virtual DOM shows the potential to reduce unnecessary interactions in real DOM. However, the current implementation of virtual DOM maintains a large virtual DOM tree ineffectively. As a possible solution, Incremental DOM might solve this issue by incrementally apply the necessary changes to the DOM tree instead of flushing and rebuilding it completely.

- The current optimisation of switching requestAnimaitonFrame to setTimeout does remarkably improve the overall performance of the framework, whereas the potential animation issue might occur when dealing with the case of running multiple visualisation instances simultaneously such as on the Data Observatory. The requestAnimaitonFrame may handle this situation in a more reliable and managed manner. Therefore, a possible solution to this concern is to build a

smart animation timeout module which is able to automatically switch and adjust the timeout functions and parameters based on the number of instances and animated elements in a visualisation.

- The pure peer-to-peer network has its advantages compared with the independent server structure in terms of overall performance and the simplicity in its components. On the other hand, the potential drawback of the current super-peer design in pure peer network may lead to the network vulnerability in case of the super-peer failure. For addressing this issue, a possible approach is to increase the super-peer redundancy in the network for reliability in the cost of performance, and we believe further research is needed in this area in order to find a more suitable or balanced solution.

- The fast iteration and development of D3.js in its very active open-sourced community have led to the issue of lacking timely support in Distributed D3. The changes and updates can be potentially useful and important for the users of Distributed D3. In order to address this development latency and have more collaborators involved in the later development, we hope the situation can be resolved or improved by publishing and creating an open-sourced community in the near future.

To make further progress on the development of Distributed D3, we believe these research topics and areas need to be addressed and covered in the next stage.

References

[1] 'Open Data - data.gov.uk'. [Online]. Available: https://data.gov.uk/. [Accessed: 17-Aug-2019].

[2] 'London Data Store – Greater London Authority'. [Online]. Available: https://data.london.gov.uk/. [Accessed: 17-Aug-2019].

[3] M. Bostock, V. Ogievetsky, and J. Heer, 'D³ Data-Driven Documents', *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.

[4] 'd3/d3: Bring data to life with SVG, Canvas and HTML.' [Online]. Available: https://github.com/d3/d3. [Accessed: 02-Sep-2019].

[5] Mike Bostock, 'Recent and archived D3.js work for The New York Times'. [Online]. Available: https://www.nytimes.com/by/mike-bostock. [Accessed: 18-Jul-2019].

[6] D. Flanagan, *JavaScript : the definitive guide*. O'Reilly, 2006.

[7] 'Apache Hadoop'. [Online]. Available: https://hadoop.apache.org/. [Accessed: 02-Sep-2019].

[8] 'Apache Spark™ - Unified Analytics Engine for Big Data'. [Online]. Available: https://spark.apache.org/. [Accessed: 02-Sep-2019].

[9] S. Whitman, 'A task adaptive parallel graphics renderer', in *Proceedings of 1993 IEEE Parallel Rendering Symposium*, pp. 27-34,.

[10] S. Molnar, D. Ellsworth, H. Fuchs, and M. Cox, 'A Sorting Classification of Parallel Rendering', *IEEE Comput. Graph. Appl.*, vol. 14, no. 4, pp. 23–32, 1994.

[11] V. Mateevitsi and B. Levy, 'Scalable Adaptive Graphics Environment: A Novel Way to View and Manipulate Whole-Slide Images', *Anal. Cell. Pathol.*, vol. 2014, pp. 1–1, Dec. 2014.

[12] G. Humphreys *et al*., 'Chromium: A stream-processing framework for interactive rendering on clusters', in *ACM Transactions on Graphics*, 2002, vol. 21, no. 3, pp. 693–702.

[13] H. Peng, H. Xiong, and J. Shi, 'Parallel-SG: Research of parallel graphics rendering system on PC-Cluster', in *Proceedings - VRCIA 2006: ACM International Conference on Virtual Reality Continuum and its Applications*, 2006, pp. 27–33.

[14] H. Chen *et al*., 'Data distribution strategies for high-resolution displays', *Comput. Graph.*, vol. 25, no. 5, pp. 811–818, Oct. 2001.

[15] T. Ni, G. S. Schmidt, O. G. Staadt, M. A. Livingston, R. Ball, and R. May, 'A survey of large high-resolution display technologies, techniques, and applications', in *Proceedings - IEEE Virtual Reality*, 2006, vol. 2006, p. 31.

[16] H. Chung, C. Andrews, and C. North, 'A survey of software frameworks for cluster-based large high-resolution displays', *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 8. IEEE Computer Society, pp. 1158–1177, 2014.

[17] L. Renambot *et al*., 'SAGE2: A collaboration portal for scalable resolution displays', *Futur. Gener. Comput. Syst.*, vol. 54, pp. 296–305, Jan. 2016.

[18] R. E. F. and K. E. Martin, 'DMX: Distributed Multi- headed X'. 2004.

[19] T. Marrinan *et al*., 'SAGE2: A New Approach for Data Intensive Collaboration Using Scalable Resolution Shared Displays', in *Proceedings of the 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2014.

[20] G. P. Johnson, G. D. Abram, B. Westing, P. Navr'til, and K. Gaither, 'DisplayCluster: An Interactive Visualization Environment for Tiled Displays', in *2012 IEEE International Conference on Cluster Computing*, 2012, pp. 239–247.

[21] K. D. Moreland, 'IceT users' guide and reference.', Albuquerque, NM, and Livermore, CA (United States), Jan. 2011.

[22] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan, 'WireGL: A scalable graphics system for clusters', in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2001*, 2001, pp. 129–140.

[23] Nirnimesh, P. Harish, and P. J. Narayanan, 'Garuda: A scalable tiled display wall using commodity PCs', in *IEEE Transactions on Visualization and Computer Graphics*, 2007, vol. 13, no. 5, pp. 864–877.

[24] S. Eilemann, M. Makhinya, and R. Pajarola, 'Equalizer: A scalable parallel rendering framework', *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 3, pp. 436–452, May 2009.

[25] K. Doerr and F. Kuester, 'CGLX: A Scalable, High-Performance Visualization Framework for Networked Display Environments', *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 3, pp. 320–332, Mar. 2011.

[26] M. Ward, G. G. Grinstein, and D. Keim, *Interactive data visualization : foundations, techniques, and applications*. A K Peters, 2010.

[27] S. Murray, *Interactive Data Visualization for the Web: AN INTRODUCTION TO DESIGNING WITH D3*. 2017.

[28] M. Aparicio and C. J. Costa, 'Data visualization', *Commun. Des. Q. Rev.*, vol. 3, no. 1, pp. 7–11, Jan. 2015.

[29] N. Bikakis, 'Big Data Visualization Tools', in *Encyclopedia of Big Data Technologies*, 2019.

[30] A. Kirk, *Data visualization : a successful design process : a structured design approach to equip you with the knowledge of how to successfully accomplish any data visualization challenge efficiently and effectively*. Packt Pub, 2012.

[31] N. P. N. Iliinsky and J. (Graphic designer) Steele, *Designing data visualizations*. O'Reilly, 2011.

[32] B. Shneiderman, 'Eyes have it: a task by data type taxonomy for information

visualizations', in *IEEE Symposium on Visual Languages, Proceedings*, 1996.

[33] J. C. Jackson, *Web technologies : a computer science perspective*. Pearson/Prentice Hall, 2007.

[34] 'Cascading Style Sheets, Level 2'. [Online]. Available: https://www.w3.org/TR/1998/REC-CSS2-19980512/. [Accessed: 03-Jun-2019].

[35] A. Rauschmayer, *Speaking JavaScript*. O'Reilly Media, 2014.

[36] 'd3/d3-selection: Transform the DOM by selecting elements and joining to data.' [Online]. Available: https://github.com/d3/d3-selection. [Accessed: 10-Jul-2019].

[37] 'd3/d3-fetch: Convenient parsing for Fetch.' [Online]. Available: https://github.com/d3/d3-fetch. [Accessed: 10-Jul-2019].

[38] 'd3/d3-scale: Encodings that map abstract data to visual representation.' [Online]. Available: https://github.com/d3/d3-scale. [Accessed: 12-Jul-2019].

[39] 'd3/d3-axis: Human-readable reference marks for scales.' [Online]. Available: https://github.com/d3/d3-axis. [Accessed: 12-Jul-2019].

[40] 'Styled Axes / D3 / Observable'. [Online]. Available: https://observablehq.com/@d3/styled-axes. [Accessed: 12-Jul-2019].

[41] 'd3/d3-transition: Animated transitions for D3 selections.' [Online]. Available: https://github.com/d3/d3-transition. [Accessed: 20-Jun-2019].

[42] 'd3/d3-timer: An efficient queue for managing thousands of concurrent animations.' [Online]. Available: https://github.com/d3/d3-timer. [Accessed: 20-Jun-2019].

[43] 'd3/d3-shape: Graphical primitives for visualization, such as lines and areas.' [Online]. Available: https://github.com/d3/d3-shape. [Accessed: 20-Jun-2019].

[44] 'Line Chart / D3 / Observable'. [Online]. Available: https://observablehq.com/@d3/line-chart. [Accessed: 06-Jul-2019].

[45] ‘Stacked Bar Chart / D3 / Observable’. [Online]. Available: https://observablehq.com/@d3/stacked-bar-chart. [Accessed: 06-Jul-2019].

[46] ‘d3/d3-color: Color spaces! RGB, HSL, Cubehelix, CIELAB, and more.’ [Online]. Available: https://github.com/d3/d3-color. [Accessed: 20-Jun-2019].

[47] ‘d3/d3-hierarchy: 2D layout algorithms for visualizing hierarchical data.’ [Online]. Available: https://github.com/d3/d3-hierarchy. [Accessed: 20-Jun-2019].

[48] ‘Tidy Tree / D3 / Observable’. [Online]. Available: https://observablehq.com/@d3/tidy-tree. [Accessed: 20-Jun-2019].

[49] ‘Treemaps for space-constrained visualization of hierarchies’. [Online]. Available: http://www.cs.umd.edu/hcil/treemap-history/. [Accessed: 30-May-2019].

[50] ‘Treemap / D3 / Observable’. [Online]. Available: https://observablehq.com/@d3/treemap. [Accessed: 20-Jun-2019].

[51] ‘Circle Packing / D3 / Observable’. [Online]. Available: https://observablehq.com/@d3/circle-packing. [Accessed: 20-Jun-2019].

[52] ‘d3/d3-geo: Geographic projections, spherical shapes and spherical trigonometry.’ [Online]. Available: https://github.com/d3/d3-geo. [Accessed: 05-Jul-2019].

[53] ‘topojson/topojson: An extension of GeoJSON that encodes topology’. [Online]. Available: https://github.com/topojson/topojson. [Accessed: 12-Jun-2019].

[54] ‘ESRI Shapefile Technical Description’. [Online]. Available: https://support.esri.com/en/white-paper/279. [Accessed: 12-Jun-2019].

[55] ‘GeoJSON’. [Online]. Available: https://geojson.org/. [Accessed: 20-Jun-2019].

[56] A. S. Tanenbaum and M. van Steen, *Distributed systems : principles and paradigms*. .

[57] K. L. Narayan, K. M. Rao, and M. M. M. Sarcar, *Computer aided design and manufacturing*. Prentice-Hall of India, 2008.

[58] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, 'A survey and comparison of peer-to-peer overlay network schemes', *IEEE Commun. Surv. Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[59] K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth, 'The Essence of P2P: A Reference Architecture for Overlay Networks', in *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pp. 11–20.

[60] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and I. Stoica, 'Looking up data in P2P systems', *Commun. ACM*, vol. 46, no. 2, p. 43, Feb. 2003.

[61] J. Risson and T. Moors, 'Survey of research towards robust peer-to-peer networks: Search methods', *Comput. Networks*, vol. 50, no. 17, pp. 3485–3521, Dec. 2006.

[62] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, 'The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations', Springer, Berlin, Heidelberg, 2004, pp. 79–98.

[63] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, 'Gossip-based peer sampling', *ACM Trans. Comput. Syst.*, vol. 25, no. 3, pp. 8-es, Aug. 2007.

[64] B. Yang and H. Garcia-Molina, 'Designing a super-peer network', in *Proceedings - International Conference on Data Engineering*, 2003.

[65] P. Garbacki, D. H. J. Epema, and M. van Steen, 'A two-level semantic caching scheme for super-peer networks', in *10th International Workshop on Web Content Caching and Distribution (WCW'05)*, 2005, pp. 47–55.

[66] B. Cohen and B. Cohen, 'Incentives Build Robustness in BitTorrent', 2003.

[67] E. Adar, E. Adar, and B. A. Huberman, 'Free Riding on Gnutella', *FIRST MONDAY*, vol. 5, p. 2000, 2000.

[68] S. Saroiu, K. P. Gummadi, and S. D. Gribble, 'Measuring and analyzing the characteristics of Napster and Gnutella hosts', *Multimed. Syst.*, vol. 9, no. 2, pp. 170–184, Aug. 2003.

[69] M. Yang, Z. Zhang, X. Li, and Y. Dai, 'An Empirical Study of Free-Riding Behavior in the Maze P2P File-Sharing System', Springer, Berlin, Heidelberg, 2005, pp. 182–192.

[70] G. Pierre and M. van Steen, 'Globule: a collaborative content delivery network', *IEEE Commun. Mag.*, vol. 44, no. 8, pp. 127–133, Aug. 2006.

[71] OMG, 'UML 2.4.1 Superstructure Specification', *October*, 2004.

[72] M. Shaw and P. Clements, 'Field guide to boxology: Preliminary classification of architectural styles for software systems', in *Proceedings - IEEE Computer Society's International Computer Software and Applications Conference*, 1997.

[73] N. R. Mehta, N. Medvidovic, and S. Phadke, 'Towards a taxonomy of software connectors', in *Proceedings of the 22nd international conference on Software engineering - ICSE '00*, 2000, pp. 178–187.

[74] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley, 2013.

[75] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, 'The many faces of publish/subscribe', *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[76] 'Data Observatory | Data Science Institute | Imperial College London'. [Online]. Available: http://www.imperial.ac.uk/data-science/data-observatory/. [Accessed: 17-Aug-2019].

[77] 'Technical Specifications | Data Science Institute | Imperial College London'. [Online]. Available: http://www.imperial.ac.uk/data-science/data-observatory/technical-specifications/. [Accessed: 17-Aug-2019].

[78] S. Manjrekar, S. Sandilya, D. Bhosale, S. Kanchi, A. Pitkar, and M. Gondhalekar, 'CAVE: An Emerging Immersive Technology -- A Review', in *2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation*, 2014, pp. 131–136.

[79] A. Febretti *et al.*, 'CAVE2: a hybrid reality environment for immersive simulation and information analysis', 2013, vol. 8649, p. 864903.

[80] 'CAVELib V3.2 | Scientific Computing World'. [Online]. Available: https://www.scientific-computing.com/press-releases/cavelib-v32. [Accessed: 12-Jun-2019].

[81] 'Industries in VR - Virtual Collaboration - TechViz Virtual Reality Software'. [Online]. Available: https://www.techviz.net/industries/. [Accessed: 17-Aug-2019].

[82] 'WebRTC Home | WebRTC'. [Online]. Available: https://webrtc.org/. [Accessed: 19-Jun-2019].

[83] 'About HTML5 WebSocket - Powered by Kaazing'. [Online]. Available: https://www.websocket.org/aboutwebsocket.html. [Accessed: 19-Jun-2019].

[84] 'websockets vs webrtc | 7 Most Amazing Comparisons To Learn'. [Online]. Available: https://www.educba.com/websockets-vs-webrtc/. [Accessed: 19-Jun-2019].

[85] 'WebGL: 2D and 3D graphics for the web - Web APIs | MDN'. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API. [Accessed: 06-Jun-2019].

[86] 'HTML5 SVG'. [Online]. Available: https://www.w3schools.com/html/html5_svg.asp. [Accessed: 06-Jun-2019].

[87] 'HTML5 Canvas'. [Online]. Available: https://www.w3schools.com/html/html5_canvas.asp. [Accessed: 06-Jun-2019].

[88] 'OpenGL ES Overview - The Khronos Group Inc'. [Online]. Available: https://www.khronos.org/opengles/. [Accessed: 09-Sep-2019].

[89] 'Large Graphs Demo [yFiles for HTML]'. [Online]. Available: https://live.yworks.com/demos/view/largegraphs/. [Accessed: 06-Jun-2019].

[90] 'D3.js - Data-Driven Documents'. [Online]. Available: https://d3js.org/. [Accessed:

18-May-2019].

[91] ‘JavaScript InfoVis Toolkit’. [Online]. Available: https://philogb.github.io/jit/. [Accessed: 12-Jun-2019].

[92] ‘Google Visualization API Reference | Charts | Google Developers’. [Online]. Available: https://developers.google.com/chart/interactive/docs/reference. [Accessed: 12-Jun-2019].

[93] ‘Springy - A force directed graph layout algorithm in JavaScript.’ [Online]. Available: http://getspringy.com/. [Accessed: 12-Jun-2019].

[94] ‘Polymaps’. [Online]. Available: http://polymaps.org/. [Accessed: 12-Jun-2019].

[95] ‘dimple - A simple charting API for d3 data visualisations’. [Online]. Available: http://dimplejs.org/. [Accessed: 12-Jun-2019].

[96] ‘Sigma js’. [Online]. Available: http://sigmajs.org/. [Accessed: 12-Jun-2019].

[97] ‘Raphaël—JavaScript Library’. [Online]. Available: https://dmitrybaranovskiy.github.io/raphael/. [Accessed: 05-Jul-2019].

[98] ‘linkorb/graphael: Graphael: GraphQL Server library’. [Online]. Available: https://github.com/linkorb/graphael. [Accessed: 12-Jun-2019].

[99] ‘Leaflet - a JavaScript library for interactive maps’. [Online]. Available: https://leafletjs.com/. [Accessed: 06-Jun-2019].

[100] ‘Create charts and maps with Datawrapper’. [Online]. Available: https://www.datawrapper.de/. [Accessed: 12-Jun-2019].

[101] ‘Flot: Attractive JavaScript plotting for jQuery’. [Online]. Available: https://www.flotcharts.org/. [Accessed: 12-Jun-2019].

[102] ‘NVD3’. [Online]. Available: http://nvd3.org/. [Accessed: 25-Jun-2019].

[103] ‘three.js – JavaScript 3D library’. [Online]. Available: https://threejs.org/. [Accessed: 25-Jun-2019].

[104] X. Fan, E. Courdier, G. Paillot, D. Birch, and Y. Guo, 'Distributed D3: A D3.js-based Distributed Data Visualisation Framework for Big Data', 2019.

[105] 'HTML DOM getBoundingClientRect() Method'. [Online]. Available: https://www.w3schools.com/jsref/met_element_getboundingclientrect.asp. [Accessed: 30-May-2019].

[106] 'The most popular database for modern apps | MongoDB'. [Online]. Available: https://www.mongodb.com/. [Accessed: 25-Jun-2019].

[107] 'OData - the Best Way to REST'. [Online]. Available: https://www.odata.org/. [Accessed: 25-Jun-2019].

[108] 'What is REST – Learn to create timeless REST APIs'. [Online]. Available: https://restfulapi.net/. [Accessed: 25-Jun-2019].

[109] 'OData JavaScript library - o.js'. [Online]. Available: https://www.odata.org/blog/OData-JavaScript-library-o.js-explained/. [Accessed: 30-May-2019].

[110] 'Real-time ASP.NET with SignalR | .NET'. [Online]. Available: https://dotnet.microsoft.com/apps/aspnet/signalr. [Accessed: 25-Jun-2019].

[111] 'PeerJS - Simple peer-to-peer with WebRTC'. [Online]. Available: https://peerjs.com/. [Accessed: 25-Jun-2019].

[112] 'Visualisation Case Studies - Data Science Institute'. [Online]. Available: http://www.imperial.ac.uk/data-science/data-observatory/visualisation-case-studies/. [Accessed: 05-Jul-2019].

[113] S. Woods, *Building touch interfaces with HTML5 : speed up your site and create amazing user experiences*. Peachpit Press, 2013.

[114] 'Virtual DOM and Internals – React'. [Online]. Available: https://reactjs.org/docs/faq-internals.html. [Accessed: 05-Jul-2019].

[115] 'React – A JavaScript library for building user interfaces'. [Online]. Available: https://reactjs.org/. [Accessed: 18-May-2019].

[116] 'Chrome DevTools | Tools for Web Developers | Google Developers'. [Online]. Available: https://developers.google.com/web/tools/chrome-devtools. [Accessed: 30-May-2019].

[117] X. Fan, G. Paillot, J. Bai, D. Birch, and Y. Guo, 'Generic DD3: A Generic Implementation of the Distributed D3 Framework with Optimisations', 2019.

[118] 'Minor GC vs Major GC vs Full GC | Plumbr – User Experience & Application Performance Monitoring'. [Online]. Available: https://plumbr.io/blog/garbage-collection/minor-gc-vs-major-gc-vs-full-gc. [Accessed: 06-Jul-2019].

[119] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, H. Payer, and G. Germany, 'Idle Time Garbage Collection Scheduling'.

[120] 'Documentation · V8'. [Online]. Available: https://v8.dev/docs. [Accessed: 30-May-2019].

[121] 'd3/CHANGES.md at master · d3/d3'. [Online]. Available: https://github.com/d3/d3/blob/master/CHANGES.md#timers-d3-timer. [Accessed: 30-May-2019].

[122] 'Release v4.0.0 · d3/d3'. [Online]. Available: https://github.com/d3/d3/releases/tag/v4.0.0. [Accessed: 30-May-2019].

[123] 'DOM Element setAttribute() Method'. [Online]. Available: https://www.w3schools.com/jsref/met_element_setattribute.asp. [Accessed: 30-May-2019].

[124] N. C. Zakas, *High performance JavaScript*. O'Reilly, 2010.

[125] S. A. Robbestad, *ReactJS blueprints : create powerful applications with ReactJS, the most popular platform for web developers today*. .

[126] ‘esbullington/react-d3: Modular React charts made with d3.js’. [Online]. Available: https://github.com/esbullington/react-d3. [Accessed: 30-May-2019].

[127] ‘AnSavvides/d3act: d3 with React’. [Online]. Available: https://github.com/AnSavvides/d3act. [Accessed: 30-May-2019].

[128] ‘Olical/react-faux-dom: DOM like structure that renders to React’. [Online]. Available: https://github.com/Olical/react-faux-dom. [Accessed: 06-Jul-2019].

[129] ‘Window setTimeout() Method’. [Online]. Available: https://www.w3schools.com/jsref/met_win_settimeout.asp. [Accessed: 06-Jul-2019].

[130] ‘window.requestAnimationFrame() - Web APIs | MDN’. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. [Accessed: 17-Jul-2019].

[131] ‘performance.now() - Web APIs | MDN’. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now. [Accessed: 12-Jun-2019].

[132] ‘binaryjs/js-binarypack’. [Online]. Available: https://github.com/binaryjs/js-binarypack. [Accessed: 06-Jul-2019].

[133] R. J. Fisman, S. S. Iyengar, E. Kamenica, and I. Simonson, ‘Gender Differences in Mate Selection: Evidence from a Speed Dating Experiment’, vol. 121, no. 2, pp. 673–697, 2006.

[134] ‘Parallel Coordinates - bl.ocks.org’. [Online]. Available: https://bl.ocks.org/jasondavies/1341281. [Accessed: 10-Jul-2019].

[135] ‘google/incremental-dom: An in-place DOM diffing library’. [Online]. Available: https://github.com/google/incremental-dom. [Accessed: 10-Jul-2019].

[136] ‘Timing control for script-based animations’. [Online]. Available: https://www.w3.org/TR/animation-timing/. [Accessed: 23-May-2019].

[137] 'SignalR/SignalR: Incredibly simple real-time web for .NET'. [Online]. Available: https://github.com/SignalR/SignalR. [Accessed: 09-Oct-2019].

[138] R. Harmes and D. Diaz, *Pro JavaScript design patterns*. Apress, 2008.

[139] 'Socket.IO'. [Online]. Available: https://socket.io/. [Accessed: 06-Jul-2019].

[140] 'socketio/socket.io: Realtime application framework (Node.JS server)'. [Online]. Available: https://github.com/socketio/socket.io. [Accessed: 20-Jun-2019].

[141] 'Node.js'. [Online]. Available: https://nodejs.org/en/. [Accessed: 06-Jul-2019].

[142] 'Express - Node.js web application framework'. [Online]. Available: https://expressjs.com/. [Accessed: 06-Jul-2019].