School of Electronic Engineering and Computer Science

Peer-to-peer update dissemination in browser-based networked virtual environments

Yousef Amar

Submitted in partial fulfillment of the requirements of the Degree of Doctor of Philosophy

2020-04-10

Statement of originality

I, Yousef Amar, confirm that the research included within this thesis is my own work or that where it has been carried out in collaboration with, or supported by others, that this is duly acknowledged below and my contribution indicated. Previously published material is also acknowledged below.

I attest that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge break any UK law, infringe any third party's copyright or other Intellectual Property Right, or contain any confidential material.

I accept that the College has the right to use plagiarism detection software to check the electronic version of the thesis.

I confirm that this thesis has not been previously submitted for the award of a degree by this or any other university.

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without the prior written consent of the author.

Yousef Amar
2020-04-10

Details of collaboration and publications:

This thesis was completed with the close advice and support of my supervisor, Dr Gareth Tyson. The datasets we analysed in §3 could not have been built without the gracious access/permission of Philipp Lenssen. Deciphering the undocumented historical iPlane dataset in §5.2.2 was only possible through the support of Professor Harsha V. Madhyastha. A list of publications published during the course of this PhD can be found in the front matter of this thesis on page iii.

Abstract

Networked Virtual Environments (NVEs) have always imposed strict requirements on architectures for update dissemination (UD). Clients must maintain views that are as synchronous and consistent as possible in order to achieve a level of user experience that is tolerable for the user.

In recent times, the web browser has become a viable platform on which to deploy these NVEs. Doing so adds another layer of challenges however. There is a distinct need for systems that adapt to these constraints and exploit the characteristics of this new context to achieve reliably high consistency between users for a range of use cases.

A promising approach is to carry forward the rich body of past research in peer-to-peer (P2P) networks and apply this to the problem of UD in NVEs under the constraints of a web browser. Making NVEs scalable through P2P networks is not a new concept, however previous work has always been either too specific to a certain kind of NVE, or made performance trade-offs that especially cannot work in a browser context. Furthermore, in previous work on P2P NVEs, UD has always taken the backseat compared to object management and distributed neighbour selection. The evaluation of these UD systems have as a result been one-dimensional and overly simplifying.

In this work, we begin by surveying past UD solutions and evaluation methodologies. We then capture NVE, browser, and network constraints, aided by the analysis of a rich dataset of NVE network traces that we have collected, and draw out key observations and challenges to develop the requirements for a feasible UD system. From there, we illustrate the design and implementation of our P2P UD system for NVEs in great detail, augmenting our system with novel architectural insights from the Software-Defined Networking (SDN) space. Finally, we evaluate our system under a range of workloads, test environments, and performance metrics to demonstrate that we have overcome these challenges, as well as compare our method to other existing methods, which we have also implemented and tested.

We hope that our contributions in research and resources (such as our taxonomies, NVE analysis, UD system, browser library, workload datasets, and a benchmarking framework) bring more structure as well as research and development opportunities to a relatively niche sub-field.

Publications

The following is a list of publications (and one patent), as well as brief summaries, published during the course of this PhD. For a list of publications before and after the PhD, as well as PDFs of all publications, please visit `http://yousefamar.com#publications`.

2019

Towards Cheap Scalable Browser Multiplayer

In Conference on Games (CoG) IEEE

> In this work, we introduce and evaluate a P2P-based method and library that aims to minimse running costs and development overhead for independent, multiplayer, browser games.

Zest: REST over ZeroMQ

In Proceedings of the 3rd Workshop on Security, Privacy and Trust in the Internet of Things, in conjunction with IEEE PERCOM

> In this paper we introduce, Zest (REST over ZeroMQ), a middleware technology in support of an Internet of Things (IoT).

2018

Providing Occupancy as a Service with Databox

In Proceedings of the 1st ACM International Workshop on Smart Cities and Fog Computing (pp. 29-34). ACM

> In this paper we present Occupancy-as-a-Service (OaaS) implemented as an app on Databox.

An Information-Theoretic Approach to Time-Series Data Privacy

In Proceedings of the 1st Workshop on Privacy by Design in Distributed Systems (p. 3). ACM

In this paper, we present a system for tuning a data consumer's access to personal data based on real-time privacy metrics.

Building Accountability into the Internet of Things: The IoT Databox Model

Journal of Reliable Intelligent Environments (pp. 1-17) Springer

This paper outlines the IoT Databox model as a means of making the Internet of Things (IoT) accountable to individuals.

2017

Balanced Message Distribution in Distributed Message Handling Systems

US Patent (serial number: 15/794440)

This patent describes a combination of methodologies for arriving at near-optimal message distribution decisions in distributed messaging systems under specific constraints.

Route-based Authorization and Discovery for Personal Data

In the 11th EuroSys Doctoral Workshop

When faced with systems in which third party components need to advertise the availability of data they gather, while other such components need to access it, solutions for delegated authorisation and discovery APIs for inter-operability are needed. This work explores possible solutions, and converges on a testable implementation.

2016

Personal Data Management with the Databox: What's Inside the Box?

In Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking (pp. 49-54) ACM

A more detailed look at the Databox as it stood; a collection of physical and cloudhosted software components that provide for an individual data subject to manage, log and audit access to their data by other parties.

Privacy-Aware Infrastructure for Managing Personal Data

In Proceedings of the 2016 ACM SIGCOMM Conference (pp. 571-572) ACM

A poster abstract giving an overview of Databox systems as they stood with a stronger focus on arbiter interactions.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Research context 3

1.3 Research objectives 4

1.4 Research contributions 5

1.5 Thesis structure 7

2 Background **8**

2.1 Overview 8

2.2 Centralisation vs decentralisation 8

2.3 The separation of the control and data planes 10

2.4 The web browser as a platform 11

2.4.1 Peer-to-peer vs client-server 13

2.5 The NVE context 15

2.5.1 Acronyms and definitions 17

2.5.2 The challenge of preventing cheating in NVEs 18

2.5.3 Other NVE challenges 21

2.5.4 NVE avatar motion mobility traces and their uses 22

2.6 The underlying network 24

2.6.1 Application-layer multicast 24

2.6.2 Network coordinate systems 25

2.6.3 Network level of detail 26

2.7 Overlay networks 26

2.7.1 A taxonomy of existing topologies 27

2.7.2 Neighbour selection metrics 33

2.7.3 Routing over P2P NVEs 34

2.8 Evaluation metrics . . . 35
2.8.1 Consistency / Staleness . . . 36
2.8.2 Bandwidth . . . 37
2.8.3 Upload/download per node . . . 38
2.8.4 Protocol Quality . . . 38
2.8.5 Delay . . . 39
2.8.6 Reliability . . . 40
2.8.7 Drift distance . . . 40
2.9 Evaluation workloads . . . 41
2.9.1 Simulation testbeds . . . 41
2.9.2 Mobility workloads . . . 41

3 Characterising Networked Virtual Environments 44
3.1 Overview . . . 44
3.1.1 Measurement objectives . . . 44
3.2 Existing datasets . . . 46
3.3 Our dataset . . . 47
3.3.1 Overview of Manyland . . . 48
3.3.2 Data collection . . . 49
3.4 Areas and workloads . . . 52
3.5 Measuring AOI density . . . 55
3.5.1 AOI density due to player activity . . . 55
3.5.2 Hotspots . . . 60
3.6 Measuring topology churn . . . 64
3.6.1 Session length . . . 64
3.6.2 Idle behaviour . . . 66
3.6.3 Motion flow . . . 72
3.6.4 Crowd behaviour . . . 77
3.7 Measuring cheating behaviour . . . 83
3.8 Measuring player reputation . . . 89
3.8.1 Vote-based rank . . . 89
3.8.2 Social network metrics . . . 92
3.9 Measuring device heterogeneity . . . 101
3.10 Measuring browser constraints . . . 104

3.11 Summary 108

4 **P2P Update Dissemination** **111**
4.1 Overview 111
4.2 Decentralisation: to what extent? 111
4.3 Architecture 114
4.4 Routing within our system 115
4.5 Our algorithm 118
4.5.1 Distance metrics used 120
4.5.2 Building an appropriate coordinate system 123
4.5.3 Computing topologies 126
4.6 Pre-connection outside of AOI 127
4.7 Mitigating cheating 128
4.8 Implementation 129
4.8.1 Signalling server 130
4.8.2 Supported topologies 133
4.8.3 Client library 137

5 **Evaluation** **141**
5.1 Overview 141
5.1.1 Evaluation questions 141
5.2 Setup and methodology 144
5.2.1 Our testbed 144
5.2.2 Modelling communication between nodes 147
5.2.3 Topologies evaluated 150
5.2.4 Workloads evaluated 150
5.2.5 Performance metrics used 153
5.3 Evaluating scalability 156
5.3.1 Performance against AOI density 156
5.3.2 Meeting browser constraints 161
5.4 Evaluating churn sensitivity 167
5.5 Evaluating packet loss resilience 171
5.6 Evaluating cheating mitigation 173
5.6.1 The effectiveness of limiting cheater influence 174

5.6.2 Cheater impact on performance 179
5.7 Summary 183

6 Conclusion **184**
6.1 Thesis summary 184
6.2 Future work 186

List of Figures

2.1 Networked Virtual Environment (NVE) gameplay screenshots across genres from "Counter Strike: Global Offensive", "Manyland", "Diep.io", and "Second Life" respectively . . . 16

2.2 A taxonomy of update dissemination topologies of major P2P architectures $*$ — hybrid Peer-to-peer (P2P) with dedicated servers . . . 28

3.1 A set of gameplay screenshots from across different Manyland areas, courtesy of the official Manyland press kit . . . 48

3.2 An empirical cumulative distribution function of total area visitors ([0, 1000] for visualisation) . . . 53

3.3 Mean Area-of-Interest (AOI) density across areas sampled at ten-minute intervals . . . 55

3.4 Mean player count across areas sampled at ten-minute intervals. The global avatar count (green) is included to illustrate the proportion of players in the two areas of focus compared to game-wide . . . 56

3.5 Density plots of player count across areas . . . 57

3.6 AOI density versus player count across environments . . . 58

3.7 Occupancy heatmaps of the *dynamic* area with cell sizes of 1×1 (top) and 10×10 (bottom) . . . 61

3.8 Occupancy heatmaps of the *static* area with cell sizes of 1×1 (top) and 10×10 (bottom) . . . 62

3.9 Density plots of player session lengths across areas, capped at the 90$^{\text{th}}$ percentile for readability (overlap of both is purple) . . . 65

3.10 Empirical cumulative distribution functions limited between $[0, 220]$m comparing areas . . . 67

3.11 CDFs of transition probabilities across players for the dynamic (left) and static (right) areas . . . 70

3.12 Mean player finite state machines for dynamic (left) and static (right) areas 71

3.13 Velocity map of the dynamic area with cell sizes of 1×1 (top) and 10×10 (bottom) where opacity is mapped to speed and colour is mapped to angle 74

3.14 Velocity map of the static area with cell sizes of 1×1 (top) and 10×10 (bottom) where opacity is mapped to speed and colour is mapped to angle 75
3.15 A visualisation of AOI radius, viewport, and interaction radius in relation to an avatar and each other 78
3.16 A visualisation of avatar grouping as computed through three separate techniques 79
3.17 A distribution of the number of groups over time across environments . . 81
3.18 Distributions (left) and histograms (right) of groups sizes over time across areas 82
3.19 A histogram of player ranks 90
3.20 A histogram of player account ages 91
3.21 A scatterplot of players binned into 100 vertical and horizontal hexagonal bins of player rank against player account age with a Generalized Additive Model trend line (red) 91
3.22 Mift network partitioned by eigenvector centrality 94
3.23 Mift network partitioned by player rank 95
3.24 A scatterplot of players binned into 100 vertical and horizontal hexagonal bins of player rank against player eigenvector centrality with a Generalized Additive Model trend line (red) 96
3.25 Mift network partitioned by age 97
3.26 Scatterplots of players binned into 100 vertical and horizontal hexagonal bins of player weighted in-degree (mifts received) against rank (left) and age (right) 98
3.27 Mift network partitioned by community / modularity class 99
3.28 Manyland.com Google analytics as of 2019-12-05 on pages, user activity, devices, and retention 102
3.29 Manyland.com Google analytics as of 2019-12-05 on user acquisition, geography, and timing 102
3.30 Manyland.com Google analytics as of 2019-12-05 on user sessions 103
3.31 The WebRTC exchange “in a complicated diagram” by MDN 105
3.32 Connection establishment time overhead against latency (left) and loss ratio (right) 106

4.1 Density plots of movement intervals (between all peers) across areas, capped at the 90$^{\text{th}}$ percentile 113
4.2 Our high-level architecture listing the roles of the signalling server 114
4.3 Life cycle of an update 117
4.4 Our peer network topology computation pipeline 120

4.5 A frame from the visualisation of different topologies under a synthetic workload. Link thicknesses correspond to link quality, red links are redundant (ours), and red peers are designated superpeers for superpeer topologies. 134
4.6 Three networking topologies of interest between servers (rectangles) and peers/clients (circles) – client-server model (left), hosted P2P (middle), and full P2P (right) 138
4.7 An example of a computed MST topology where peers with better connections (② and ③) act as supernodes, and with redundancy (① and ④) 139

5.1 An overview of our evaluation testbed setup 145
5.2 A visualisation of the iPlane network before completely connecting (left) and a corresponding map of known/valid Point of Presence (PoP) coordinates (right) 148
5.3 A breakdown of the synthetic trace payload 152
5.4 Illustration of position drift, missing peers, and extra peers, from the perspective of the dark blue player and their grey AOI 154
5.5 Mean drift distance distribution across real environment traces; dynamic (left) and static (right) 157
5.6 Mean missing ratio distribution across real environment traces; dynamic (left) and static (right) 158
5.7 Mean extra ratio distribution across real environment traces; dynamic (left) and static (right) 158
5.8 Mean mean drift distance across the high-churn synthetic workload for different numbers of players 160
5.9 Mean missing (left) and extra (right) ratio distributions across the high-churn synthetic workload 160
5.10 Maximum active degree by number of peers over the synthetic workload . 162
5.11 Mean upload (left) and download (right) rate distributions across the dynamic environment workload ordered by mean 164
5.12 Mean upload (left) and download (right) rate distributions across the static environment workload ordered by mean 165
5.13 Mean upload (top) and download (bottom) rates across the synthetic workload for different numbers of players 166
5.14 Proportion of updates lost due to cooldown across dynamic (top) and static (bottom) workloads 168
5.15 Mean ratio of updates dropped due to cooldown over the synthetic workload for different levels of churn 170
5.16 Mean drift distance under a range of artificially induced loss ratios 172

5.17 Illustration of cheater peer (red) influence on the propagation of updates from a source peer (blue) in networks across a spectrum from low to high cheater influence 174
5.18 Mean normalised cheater betweenness measures for different cheater probabilities across the dynamic (top) and static (bottom) workloads 176
5.19 Worst case drift distances for different cheater probabilities across dynamic (top) and static (bottom) workloads 180

List of Tables

2.1 Summary of latency and online games from [26] 17
2.2 Acronyms in the VE space 18

3.1 Statistical summary of player count distributions across environments . . 57
3.2 Additional occupancy heatmap information 63
3.3 Statistical summary of idle time distributions across environments 68
3.4 Additional velocity map information 76
3.5 Summary of distributions of the number of groups over time across areas 81
3.6 Summary of mift network statistics 100

5.1 Summary of evaluation results with best results highlighted green, acceptable yellow, and worst red 143
5.2 Table of standard deviations of mean normalised cheater betweenness measures for every topology (rows) and cheater probability (columns) across the dynamic workload 177
5.3 Table of standard deviations of mean normalised cheater betweenness measures for every topology (rows) and cheater probability (columns) across the static workload 177
5.4 Table of standard deviations of worst-case drift distances for every topology (rows) and cheater probability (columns) across the dynamic workload . 181
5.5 Table of standard deviations of worst-case drift distances for every topology (rows) and cheater probability (columns) across the static workload . . . 181

Chapter 1

Introduction

1.1 Motivation

The internet, and the economies it creates, are volatile and unpredictable. Within our lifetimes, reality has become deeply permeated with digital worlds upon worlds, for better or for worse. Change at this rate and scale is disruptive — a large part of the population is older than the internet itself and everything it enabled.

As the internet landscape changes, unprecedented systems and system architectures can suddenly become possible. At the network level, virtualisation has changed the status quo in the past decade, in the form of Software-Defined Networking (SDN). Taking networking functions that were traditionally coupled to physical hardware, and abstracting them away into software, has enabled a level of flexibility in managing network infrastructure that would have been previously mired in complexity. The key SDN tenet of separating the data plane from the control plane, and logically centralising the latter, has yet to be applied to many other networking contexts.

In recent years, we have also seen a trend towards edge computing and, more generally, shifting computation away from servers in datacenters and towards clients. This can be attributed to many factors, the most prominent being the rise in the capabilities and ubiquitousness of smartphones and other personal devices. While this makes edge computing more viable, as a result of these devices the world has seen an explosion in the quantity of personal data people produce daily, in the past decade alone [68]. This in turn reinforces this shift through legal, privacy, and latency requirements.

Latency requirements are especially prominent in Internet of Things (IoT), Virtual Reality (VR), and Networked Virtual Environment (NVE) use cases, among others. Ex-

amples of NVEs are multiplayer games, which need to achieve a level of synchronization and consistency that is acceptable to the users or risk detriment to user experience.

At the same time, desktop applications are being eclipsed by web applications to the point where many can have their needs met completely with just a web browser. Treating the web browser as an operating system creates its own challenges. Recent standards and browser developments, such as WebAssembly, WebGL, WebVR, WebRTC, and WebUSB, have made the browser rife as a deployment platform for interactive applications, networked VR, and high-end games. However, the default approach is to use a sever-based model to synchronise clients, which has implications for scalability, performance, and fault-tolerance.

With this recent push towards client-side processing and decentralisation, Peer-to-peer (P2P) networks are often considered as an approach to decentralising the communication between clients too. However, in order for P2P networks to be a viable alternative to client-server architectures for NVEs with low latency requirements (such as multiplayer games), P2P topologies need to optimise for different performance metrics while remaining scalable as the number of peers increases. More specifically, an approach for building an overlay network for update dissemination is needed.

Our goal is to therefore build a system that enables P2P update dissemination in a way that can be tuned to different NVE use cases, that can also be used in a web browser context. We approach this goal under the precept that building P2P topologies for this use case (the control plane) need not be conflated with update dissemination (the data plane), and that in fact the logical separation of the two can yield previously untapped benefits. A viable system like this also yields further high-level benefits:

- The correlation between server costs and number of users would become much weaker, leveling the playing field for independent developers and small startups, as it would allow them to build scalable applications without the financial barrier to entry.

- P2P, especially in the browser, is very technically complex. Scalable P2P for real-time applications has not been attempted successfully in the browser at all. Encapsulating this complexity in libraries and APIs would further the development barrier even further.

- P2P update dissemination has the potential to improve the performance (latency etc) of online applications especially when the providers cannot afford distributing servers across continents.

- P2P architectures for real-time applications have the potential to be more fault-tolerant than traditional client-server architectures, and more resilient to viral spikes in load, if designed right.

A system like this must also be compared to other P2P systems to show under which conditions it will outperform them. In order to do this, we must implement a range of existing solutions and categorically representative solutions alongside our own. While there are many frameworks for evaluating general-purpose P2P networks, there are virtually none for our specific use case. A framework for testing and comparing these systems and topologies would therefore be a valuable contribution to the field, and so we build such a framework and use it to compare our method with others along the main performance metrics that are most prevalent in the literature.

1.2 Research context

We focus specifically on P2P update dissemination for NVEs in a browser context. P2P networks for NVEs are rare, but exist, normally focusing on object management, which we explore in the next chapter. Update dissemination in P2P NVEs is an even smaller subset of this literature, despite it arguably being the most important by sheer volume of network traffic. Doing this under the constraints of browsers is completely unprecedented.

We do think it is important to explore existing techniques however, despite us having tighter constraints. We explore existing approaches, both structured and unstructured, in detail in the next chapter. Many of these (especially structured networks based on Distributed Hash Tables (DHTs)) are general-purpose P2P solutions that do not exploit the rich application-layer data that can be used to improve topologies, such as virtual avatar positions and areas of interest. Other solutions do take account of this data and use locality-sensitive hashing (in DHT-based approaches) or other techniques such as Delaunay triangulation over virtual positions to compute an unstructured network.

Often, these approaches reach the extreme on the other end, in that they are static and suited to very specific use cases, or e.g. genres of games, to eke out better performance through specialisation at the design stage. Further, these systems often base their results on oversimplified models of real networks, ignoring factors such as heterogeneous quality of links between nodes, packet loss, fault-tolerance, and node bandwidth capacities. This is especially pronounced in the P2P space as the resources available on devices vary much more significantly than hosts in a datacenter.

Furthermore, there is an unfilled need for scalable systems such as these in the context

of web browsers — a context we map extensively in this thesis. The browser imposes additional constraints, such as the overhead in creating a P2P connection, and limits to the number of concurrent connections; practical constraints that theoretical/simulated previous work does not take into account.

NVEs come in all shapes and sizes; from fast-paced First-Person Shooters (FPSs), to social Massively Multiplayer Online Games (MMOGs). We focus on developing a method for computing update dissemination topologies for a range of different NVEs. The fundamental goal of update dissemination is to give peers a consistent view of the avatars within their areas of interest. In a perfect world, peers could communicate instantly with unlimited bandwidth, resulting in perfect consistency. The reality is that network conditions create imperfect consistency, however as long as the consistency is close enough, this is acceptable (or not noticeable) to players.

It is very standard that NVEs do some level of position interpolation and prediction in order to increase consistency. This is out of our scope, and can be applied to any system. Naturally, the predictions are better with less stale inputs. Similarly, cloud-based decentralisation is out of our scope, as the motivation of our work applies specifically to the advantages of P2P update dissemination, although throughout this thesis we will continuously compare different approaches to a standard client-server architecture as a baseline.

1.3 Research objectives

Our top level goal is to build a system that enables **P2P update dissemination** in a way that can be **tuned to different NVE use cases**, and that can be used under the constraints of a **web browser context**. We can further decompose this goal into three parts that roughly correspond to one chapter each, after our background chapter.

1. To capture NVE, browser, and network requirements for P2P Update Dissemination (UD) systems
2. To design and implement a P2P UD system that meets these requirements
3. To evaluate our implementation and demonstrate that it does indeed meet these requirements with respect to alternative solutions

These objectives are difficult as existing systems can meet only a small subset of these requirements. UD in NVEs is not a very deeply researched topic as is, but the few

existing techniques tend to sacrifice versatility in favour of high performance for specific applications. By focusing on NVE update dissemination in this context, we aim to create a system that is versatile enough to meet these requirements while still scoring high on typical performance metrics. Our design is directly driven by the requirements we capture, making it better suited to meeting these, unlike other approaches.

1.4 Research contributions

This section summarises the major contributions presented in this thesis. Each of the top-level outcomes map to a content chapter. For a list of papers published over the course of this PhD, please see the publications listing in the front matter of this thesis.

1. §2 — Background
   (a) A review of different NVEs, NVE genres, and models of these in literature
   (b) A taxonomy of different update dissemination topologies in literature
   (c) A thorough survey of performance and evaluation metrics for P2P NVEs
2. §3 — Characterising Networked Virtual Environments
   (a) Development and release of bots and crawlers for the collection of rich application-layer NVE network traces for a specific NVE
   (b) Collection and release of several months (still continuously collecting) of such traces across different genres of environments. These can be used for a wide range of analytics by the research community, not just in designing P2P networks (e.g. mobility patterns, crowd behaviour, anonymised chat logs, social network analysis)
   (c) Requirements capture for different genres of NVEs on the basis of these traces in the areas of:
      - Player activity and density patterns
      - Session churn, idle behaviour, motion flow, and crowd behavior (where we present two novel methods for group detection from NVEs mobility traces)
      - Cheating and using reputation metrics to mitigate different forms of cheating
   (d) Discovery and disclosure of Cross-Site Scripting (XSS) vulnerabilities and other exploits within this NVE

(e) Requirements capture for browsers, devices, and networks

3. §4 — P2P Update Dissemination

    (a) Design and implementation of a P2P update dissemination system tuned for browser-based NVEs

        - A novel, extensible approach to neighbour selection is incorporated. Application-layer metrics such as virtual locality, network conditions, reputation measures, and longevity, are fused and used to build a semantic coordinate space
        - Configurable guaranteed minimum connectivity is included by design, creating redundant paths for resilience to churn, loss, and cheating
        - Intelligent pre-connecting to peers to effectively mitigate browser constraints
        - Design and implementation of low-overhead cryptographic solutions to prevent tampering with forwarded update traffic
        - A generalised design that is highly configurable to different deployment environments and genres of NVE
        - Game-specific optimisations (such as delta-coding and position interpolation/prediction) are included out of the box for this context

    (b) Release of a browser library and signalling server, that supports our system and a range of other topologies, for developers to actually use[1]

4. §5 — Evaluation

    (a) A framework for the meticulous evaluation of P2P NVEs

        - Underlay networks and deployment environment constraints are realistically emulated
        - Highly parallel execution for tight experiment turnaround times
        - Support for using synthetic and trace datasets (built above) as evaluation workloads
        - Rich statistics are collected to evaluate performance in different dimensions (e.g. scalability, churn sensitivity, loss resilience, cheating mitigation)
        - Configurable to different deployment environments (e.g. browser vs general-purpose)

    (b) A full evaluation of our system, compared to other systems, using this framework

[1] `https://libfabric.com/`

1.5 Thesis structure

So far, we have described the research objectives we aim to solve at a high level, and motivated why this is needed. In this section, we briefly lay out the structure of the remainder of this thesis.

In §2 we review the literature, and explain everything that someone not specialised in this area would need to know to understand the bigger picture and where our solution fits in it. Here, we also discuss what other solutions are currently available, as well as by what metrics these systems can be compared.

We then focus on capturing the constraints and requirements within this context in §3. To do this, we build a large dataset from an existing NVE, and make targeted measurements over this and other collected data. We also run a number of experiments and analyses to understand browser and network requirements.

With these results, we describe our system design and implementation in §4. Our design is explicitly guided by our earlier conclusions as well as top-level goals.

Finally, we demonstrate that our system meets these constraints and requirements by evaluating it in both synthetic as well as real environments, over a range of parameters and workloads. Further, we show how our system compares against alternative solutions and solution classes.

We end this thesis with a summary of our work and a conclusion of our contributions and findings, as well as future work.

We recommend that this thesis is read digitally, with a PDF reader that highlights links and has navigation key bindings, for example `mupdf`. We heavily utilise cross-referencing sections in this document, so it is convenient to be able to click a reference and jump back when done. The same holds true for glossary terms, citations, URLs, and figure/table/equation references. Virtually all figures in this document are vector graphics, meaning they can be zoomed into infinitely; a capability which is lost by printing this thesis on paper. We hope that our endeavours to make this thesis a smooth read are effective.

Chapter 2

Background

2.1 Overview

In this chapter we explore and review high-level concepts and literature pertinent to the research areas of this PhD project. We begin by looking at the history of decentralisation and the browser as a platform in order to set the scene. Then, we outline the challenges of Networked Virtual Environments (NVEs), and the research that is done in this area, to frame our context. We examine more detailed underlay and overlay network research — including presenting our taxonomy of existing overlay network topologies — such that our contributions can be positioned in this bigger picture of related work. Finally, we end this chapter with a thorough survey of evaluation metrics and a discussion of evaluation workloads, to justify our later evaluation setup and methodology.

2.2 Centralisation vs decentralisation

In the sixties, when a computer with the most basic capabilities could fill an entire room, all computation was done on mainframes. People would connect to these through terminals that did nothing but interface with the mainframes, and jobs would be scheduled and run on the mainframes with shared resources. Computing resources and performance were major bottlenecks, and security was barely a concern at this point in time. This was akin to having processing concentrated server-side with minimal clients interfacing with the servers.

Once desktop computers started becoming practical and affordable in the seventies, we saw a shift away from the centralisation of mainframes, and programs were instead run

locally. Once the internet started to gain momentum, it looked much more decentralised and federated than it does today. In its early days, when IPv4 addresses were thought to never run out, and NATs were not needed, web pages were being served from the same machines that requested them.

When Napster was released in 1999, many Peer-to-peer (P2P) protocols and networks followed, and we also started seeing the internet being used for applications such as VoIP. At the same time, certain web pages were gaining a lot of traction and introducing information asymmetry on the internet. Websites were, and still are, hosted on dedicated servers that have the resources to handle larger volumes of traffic.

As servers and storage became cheaper at scale, many web service providers have adopted the model of server-side rendering, leaving client devices to mostly display static pages that would be considered simple by today's standards. Through competition, client devices and browser quickly caught up in resources and capabilities however, so modern web apps have shifted back to Client-Side Rendering (CSR), through Single-Page Applications (SPAs) and Progressive Web Apps (PWAs), in order to improve user experience and lower server costs.

More recently, there has been a push to reduce overheads and improve scalability by attempting to split application logic into microservices and more discrete units of logic [58]. Server-side deployment has gone from virtual machines, to containers which share an OS, to lambdas (in a serverless context) which share a runtime [59]. The additional granularity has the advantage of lowering server costs as you only pay for exactly what you use.

That being said, each paradigm still requires elaborate and labour-intensive orchestration and management of infrastructure at scale. Further, bandwidth costs especially can have a damning impact on the feasibility of the use cases most relevant to us, as in the worst cases these variable costs can scale quadratically with respect to the number of users. We can lower these costs significantly by shifting as much server-side logic as possible to the clients and at the same time simplify the development overhead using appropriate abstractions and as all development becomes client-side development.

The current state-of-the-art in serverless computing are Cloudflare Workers [12] which, in addition to runtime optimisations, run on servers as close to the user as possible. Even here, the pull of distribution and decentralisation is evident and strong, making a resurgence whenever it is possible, simply because the advantages are too large to ignore. To this day, new developments that utilise distributed technology, such as Distributed Hash Tables (DHTs) or blockchains, continue to create systems with advantage in resilience, privacy, scalability, cost, and more.

2.3 The separation of the control and data planes

The *Separation of Concerns* is an effective philosophy to follow in engineering. In the past decade, Software-Defined Networking (SDN) has transformed networking from the simple premise of separating the data plane from the control plane, and logically centralising the latter. Taking networking functions that were traditionally coupled to physical hardware, and abstracting them away into software, has enabled a level of flexibility in managing network infrastructure that would have been previously mired in complexity.

Virtualisation as a concept is hardly new of course and has always been prevalent in the cloud computing space especially. Before SDN, Storage Area Networks (SANs) were abstracting away the storage layer to create software-defined storage. The same philosophy has never truly been applied to P2P protocols, simply because most of these were for use cases such as file sharing, where any sort of centralisation creates intractable points of failure, censorship, or takedown. These also did not have the problem of the control plane being coupled to physical hardware, and these networks were simple enough that the control plane could be completely distributed and built into the protocol itself.

Early P2P networks were not without their problems however. When unstructured, gossip-like protocols used to be prevalent [31], information would propagate through potentially disconnected networks through different forms of flooding. These protocols could not make strong guarantees on the availability of data. This changed once structured protocols and DHTs became more common and their design could indeed make stronger guarantees. Besides the fact that these protocols generally assume heterogeneous networks — a problem we discuss DHTs in the context of NVEs in more detail in §2.7.1.1 — their control plane is also logically distributed across all nodes, through routing tables that each node maintains.

When we start looking at the NVEs use case however, we draw the critical insight that there is no reason for the control plane to be distributed at all, and to have to deal with all the challenges that brings. Control messages would make up a negligible amount of network traffic next to the data plane, i.e. in-game state dissemination. Many systems try to distribute everything for the sheer sake of distribution and being able to boast a completely distributed system, when in fact centralising the brains of P2P topology computation can even be a competitive advantage, as hiding that logic server-side as a "trade secret" allows hosts to monetise these networks.

In SDN terms, we can treat peers as forwarding devices and the game server as an SDN controller. Bringing this approach to the NVE context is promising uncharted territory and a central theme throughout our work.

2.4 The web browser as a platform

The furthest reaches of the "edge" can include smartphones, PCs, Internet of Things (IoT) hubs, even embedded devices. Edge computing deployments will only take advantage of a subset of these however, as the heterogeneity between these devices includes runtime environments. There is however a runtime environment that has reached extensive device coverage: the web browser. Even headless devices can often still run browsers in a headless mode, or run Google Chrome's V8 engine, which powers Node.js for server-side JavaScript.

The value in browser dominance has been widely recognised. As of today, Google Chrome continues to hold the majority market share on the battlefield of the browser wars [122, 121]. At the same time, browsers (at least the one adhering to modern standards) are becoming more and more capable. It has gotten to the point where a browser is able to replace most, if not all, equivalent desktop applications. Google's Chrome OS goes as far as extending the Chrome browser to become an operating system in itself, with web apps replacing desktop apps.

With users spending the majority of their computing time in the browser, this is not a surprising development, and with browsers building cross-platform APIs for applications that require lower-level hardware access (such as WebGL, WebVR, and recently, WebUSB), soon there will be very little that cannot be done within the framework of a browser. Indeed, this gives developers a different way of thinking about software development that is more dynamic, standards-driven, and OS-agnostic.

From the other side, web development has also changed to adapt to this shifting environment. With most users worldwide browsing the web through smartphones [123], especially in developing countries where a smartphone is a much smaller investment for connectivity, a "mobile-first" approach to web design has been widely adopted by developers and modern web frameworks. The standards have shifted to reflect this too, for example through the CSS3 standard with media queries and similar.

Meanwhile, as devices have been getting more powerful, software design patterns have changed to reflect that. SPAs, and more generally, CSR has become prevalent and standard, limiting the back-end to Representational State Transfer (REST) APIs and minimising interaction with servers. Web frameworks that enable this, such as React, Angular, Vue.js have soared in popularity [72] and front-end employers now usually expect familiarity with one or more of these frameworks.

This also means a greater emphasis on client-side security, the brunt of which is handled by web standards and, by extension, browser vendors. APIs such as Content

Security Policy (CSP) are available to developers to prevent Cross-Site Scripting (XSS) and attacks in the same class or similar.

Web apps have also infiltrated the desktop space. This was originally through frameworks such NW.js (previously node-webkit) and Electron, but has now become a standard with PWAs. PWAs also allow web apps to become mobile apps, a feat which was previously only possible by wrapping your web app in a native WebView object, or porting a web app to an equivalent of that with software such as Cordova or PhoneGap. PWAs introduce a whole new way of thinking about and writing web apps, that employ service workers to cache resources and enable apps to be usable offline too. Popular apps are already secretly web apps, and it is not inconceivable that the majority of desktop and mobile apps will be the same in the future.

Earlier we have explored the serverless paradigm, where the trend in cloud computing is tending towards smaller, more granular microservices with shared overheads. This trend has strong parallels at the client-side. The aforementioned shrinkage in server-side processing, and bare REST APIs that map closely to Create, Read, Update, and Delete (CRUD) operations on databases, go hand in hand with the trends we have discussed in this section.

The critical issue here is that while the back-end has been getting leaner, the front-end has seen a lot of bloat and frustration by developers — the rate at which new front-end JavaScript frameworks have been spawning is impossible for most to keep up with and causing a lot of fragmentation. Meanwhile, the very concept of SPAs intuitively does not mesh well with sane design practices employed when creating microservices, such as the *Separation of Concerns* or the *Single Responsibility Principle*.

Despite advances in serverless computing and similar areas, many application providers still rely on older technology however and are locked in to legacy architecture due to high technical debt. The transition to cheaper and more efficient architectures is therefore slow. As it stands, the standard approach for deploying a web app is to rent a Virtual Private Server (VPS), often with one of the big cloud providers such as Amazon Web Services (AWS). Most of these have tools for handling scaling and load balancing out of the box and host from big datacenters in well served locations.

A quarter of a century ago, the web was largely made up of static pages. With the advent of PHP and similar, server-side processing spawned a large range of new web app functionality. To this day, some degree of server-side processing is all but expected. PHP continues to dominate at 79% use by all websites with a known server-side programming language, while static files sit at 2.1% [135].

Today, large scale web application providers employ a multitude of optimisations in order to keep their service performant. The use of Content Delivery Networks (CDNs) is an example of one such optimisation, where web application providers may deploy or rent caching proxy servers that are geographically closer to their users in order to deliver resources such as static assets to them quicker. Some web application providers, especially those active in areas with low latency requirements such as real-time multiplayer games, go as far as operating private networks [100, 101, 102].

2.4.1 Peer-to-peer vs client-server

The most common approach to supporting multiuser virtual environments is a client-server model, where one or more authoritative servers maintain communication between clients. This holds true for both browser and non-browser games. This is usually (at least at first) the least complex solution and fits well within existing internet architecture and paradigms. The server can be fully authoritative, handling all logic and simulation (e.g. the physics engine), and the clients act as dumb terminals responsible only for sending keyboard and mouse events to the server and rendering the environment. This is usually the preferred model for slower environments that also want to make cheating, by manipulating the game state client-side, impossible.

On the other end of the spectrum are servers that are only responsible for forwarding messages between players to synchronise state. Clients handle all of their own computation, and only inform the server of state changes such as avatar positions. This is much less secure in general as a client can manipulate the client's state in many different ways (e.g. by disabling physics engine collisions or spoofing their positions etc) and therefore additional client-side security is needed.

This raises the question: why forward this data through a central server in the first place? If clients could communicate with each other directly P2P in a scalable manner, the application provider could forgo servers almost entirely and associated costs. The same challenges as above are prevalent in P2P-networked environments, as peers communicate with other peers directly, and there is no authoritative server to validate states.

The potential of P2P architectures for game networking has always been recognised, however the obvious main challenge is scaling — for every additional user, the number of connections to naively keep all users connected goes up quadratically. That being said, if this challenge can be overcome, P2P architectures have the potential to lower costs for game developers and provide a faster, more reliable service to the users. Historically, developers that have recognised the potential and attempted to overcome these challenges

have given up due to the architectural [86] and development [97] complexity.

The web browser as a target platform for modern games has been unpopular due to its limitations, for example lack of multithreading support for a single page. In recent times, developers have been deploying less and less to the browser [42, 43]. However, as modern HTML5 APIs such as WebVR, WebGL, and WebAssembly are beginning to see widespread support in browsers and devices, there is an opportunity for independent game developers to rediscover the browser as a serious target for deployment. As independent developers are more limited when it comes to labour and resources, many shy away from large-scale, real-time multiplayer game development. We know that there is a strong demand however, as for example small browser multiplayer games called "io games" (as they use the .io top-level domain) have seen a meteoric rise in popularity.

Despite this, rather than deal with financial and latency challenges, many developers opt to populate their games with fake bots to make them seem multiplayer, when in many cases they are actually completely offline. While this phenomenon has been noticed in the past by players [103, 73], we have independently verified it by monitoring network traffic (or rather, lack thereof) through Wireshark while running popular io games such as hole.io[1] and paper.io[2]. In fact, simply disconnecting from the internet mid-game, and observing that other players magically continue to move, is evidence against the game developers' claims that they do not employ the use of bots [134].

In the past, there have been many non-browser games that employ peer-to-peer (P2P) communication. As most clients/peers tend to be behind NATs, and port-forwarding is often a non-trivial ask of players, these games rely on techniques such as UDP hole punching, or supplement communication with intermediate servers. For the purpose of our research, low-level logistical challenges in setting up P2P connections are out of scope. These challenges are all but solved, and wherever it is not possible to establish a direct connection between two peers, workarounds with proxies or relay servers exist.

Client-server Virtual Environments (VEs) often have a notion of "rooms" or distinct zones, where the players within are connected to each other, but not to players in other rooms. In some VEs, different servers can host the same areas for different groups of people, to limit the number of people one one server at a time. This started in 1997 with Ultima Online calling these parallel universe-like servers "shards" to match in-game lore. This is also why the term "shards" is used to describe horizontal partitioning in databases and some distributed systems.

This same concept of disjoint rooms/shards can also map to P2P architectures. In

[1] `https://hole-io.com/`
[2] `https://paper-io.com/`

many use cases, such as First-Person Shooters (FPSs), only a small group of players need to be connected, and one peer in a group of peers is designated “host” and acts as a de facto authoritative server. This limits the server costs of the game provider to simply acting as a lobby/directory for finding these rooms/groups, but at the same time, the number of players that a host can support is more limited than a standalone server. This is a critical limitation when it comes to implementing the same for games with many players that have been recently becoming more popular, such as the Battle Royale or Massively Multiplayer Online Game (MMOG) genres, which we aim to address.

2.5 The NVE context

The different web app paradigms we touched on in the previous section are all worthy of detailed research in themselves, and depend very much on the application in question. For example, social media platforms may need their different functionalities split up in different ways where standard CSR and REST simply are not enough. These requirements literally drive the development of the currently most popular front-end framework, Facebook’s React, and others like it.

Meanwhile, online media-services providers like Netflix, that serve a large volume of static data, instead optimise through compression and replication. In these cases, infrastructure is more important than page load times for example.

Minimising latency is however becoming a much more prominent focus than throughput. Aside established areas like financial trading, we are seeing emerging technology with critical low latency requirements, such as Virtual Reality (VR) and some IoT use cases such as self-driving vehicles. To meet these requirements are solutions at the protocol-level (e.g. HTTP/2) as well as infrastructure-level (e.g. 5G and satellite constellation networks).

Throughout this thesis, we look at one class of use cases that is on the highest end of the latency requirements spectrum: real-time interactive applications, such as multiplayer games and other virtual environments. Other use cases such as collaborative editing platforms pose similar, though less pronounced, challenges.

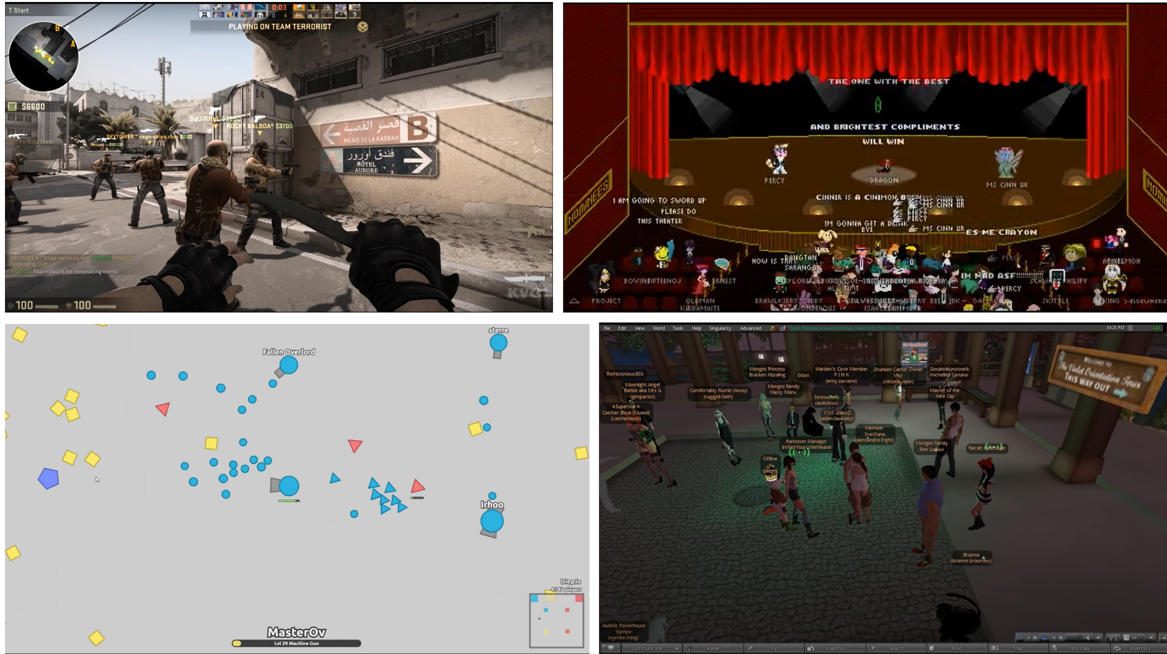


Figure 2.1: NVE gameplay screenshots across genres from “Counter Strike: Global Offensive”, “Manyland”, “Diep.io”, and “Second Life” respectively

Figure 2.1 shows gameplay screenshots we took from a range of NVEs, including one from Manyland (top right) which we later examine, as well as Second Life (bottom right) which is commonly the target of NVE research. We focus on this area because it is an extreme case that amplifies all the challenges with other use cases. This is because:

- Latency requirements are very high compared to other use cases
- Users/connections are volatile and ephemeral
- Server-side logic is usually monolithic and complicated (e.g. physics simulations)
- Because of this, costs are much more significant and are difficult to cut down through porting to e.g. a P2P architecture. This increases the developer barrier to entry
- There are strong incentives for bad behaviour (e.g. spoofing game state) so client-side processing cannot be trusted
- Interaction between users can be innumerable and simultaneous
- Different NVE genres can have significantly stricter or laxer requirements

Fundamentally, the goal of networking users in an NVE is to synchronise their views with as little lag as possible consistently. Many have investigated player latency tolerance

Model	Perspective	Example Genres	Sensitivity	Thresholds
Avatar	First Person	FPS, Racing	High	100 milliseconds
	Third Person	Sports, RPG	Medium	500 milliseconds
Omnipresent	Varies	RTS, Sim	Low	1000 milliseconds

Table 2.1: Summary of latency and online games from [26]

in the past; table 2.1 shows a summary of these acceptable latency thresholds from one such work [26] (with some variation depending on the exact game). These challenges are most evident for fast-paced multiplayer games, such as FPSs, Role-Playing Games (RPGs), as well as some Real-Time Strategy (RTS) games, and generally speaking, any latency above 500ms is unacceptable.

This area is especially challenging because the development of the architecture of the internet was not driven with this use case in mind. It is therefore important to drop all assumptions from the start and consider different network architectures and systems when searching for a solution.

2.5.1 Acronyms and definitions

Within this field of research, there are also a number of commonly used terms and acronyms, especially in a P2P context. Here, we list the most common of these.

When referring to virtual environments with many interacting users, such as MMOGs, the literature uses scattered terminology. In this work, we use the all-encompassing term VE, or NVE if the context requires that we differentiate this from a non-networked VE. Table 2.2 lists other terminology used in the literature.

Throughout this work, we may refer to the players in an NVEs as players, avatars, peers, nodes, clients, or hosts interchangeably. We disambiguate in cases where they do not mean the same thing (e.g. where a physical host might host multiple peers or avatars) but what these terms refer to is usually the same thing. When they do not, this is usually clear from the context (e.g. where a *client* in a client-server context is not a peer, or where *player* refers to a human user as opposed to their avatar).

The area around an avatar in a virtual environment is commonly called the Area-of-Interest (AOI) [118, 85] in contemporary literature. In earlier work this has also been referred to as an *awareness area* [80], *aura* [51], *Domain of Interest* [105], and *aura nimbus* [6]. The act of limiting the information that individual players have access to based on their AOI, for reasons of increasing scalability and/or decreasing cheating, is referred to as Interest Management (IM) [6, 105].

Table 2.2: Acronyms in the VE space

Acronym	Expansion
VE	Virtual Environment
VW	Virtual World
DVE	Distributed Virtual Environment
NVE	Networked Virtual Environment
CVE	Collaborative Virtual Environment
MMVE	Massively MultiUser Virtual Environment
SVW	Social Virtual World
DIA	Distributed Interactive Application
DIS	Distributed Interactive Simulation
MMG	Massively Multiplayer Game
MMOG	Massively Multiplayer Online Game

IM is generally divided into *spatial* techniques where players move across a continuous world and IM is computed dynamically, and *regional* techniques where the VE can be split into discrete zones that players can move between. In this work, we focus primarily on spatial techniques, as they are a more general form of the latter.

2.5.2 The challenge of preventing cheating in NVEs

A challenge that comes up again and again in P2P literature is how to deal with malicious behaviour. In the VE subset of this body of research, malicious behaviour usually manifests itself in the form of cheating in order to gain an advantage in-game. This challenge must be addressed as it is often one of the main arguments against a P2P architecture for NVEs.

It is important to note that some kinds of cheating will exist irrespective of the architecture. Gauthierdickey et al. [47] have organised the different types of cheating into a useful taxonomy. These types can be split into four layers: game, application, protocol (which the authors focus on), and network.

Game-layer cheats are where players break the rules of a game. An example they give in a later paper [46] is looking at someone else's hand in poker, but VE examples may include taking advantage of unintended glitches, broken game mechanics/design, or players colluding. This also includes "boosting" or "win trading" where higher-level accounts in a competitive game intentionally lose against lower-level accounts (that either they own themselves, or a friend, or somebody who paid them) in order to level them

up. These are usually combated either through patching broken mechanics or some form of behavioural analysis coupled with banning cheaters. This kind of cheating is largely irrelevant here because it is dependent on game design and will persist regardless of the type of architecture.

Network-level cheats (e.g. denial-of-service attacks) are relevant only in that P2P networks expose the IP addresses of individual players, which makes them a target, as opposed to the game server. Protocols for anonymity (be it onion routing or otherwise) are not worth the overhead in this context. For the most part, this is an acceptable downside, and modern routers have built-in measures against many of these attacks.

Application-layer cheating, where players modify or augment their game client, can also pose problems in architecture-agnostic P2P ways. There is a whole menagerie of different kinds of cheats that can be done in this manner. Common kind are "aimbots" or "trigger bots" in FPSs where a player may use an external program or modified client to improve or perfect their aim, giving them an unfair advantage. Similarly, creating macros to trigger clicks and key presses at inhuman speeds is common. Some are quite advanced at still seeming human, creating an arms-race between cheating and cheating detection, even utilising machine-learning approaches.

Another common kind of application-layer cheating is "world hacking" or "maphacks" where clients have full knowledge of the game environment and can remove objects that make the game more challenging, such as fog-of-war in RTS games. This extends to removing visual elements, such as particle effects or camera shake (e.g. gun recoil) to make the game easier. A subset of this is the "wallhack" where cheaters modify their clients such that they can see other players through walls to anticipate them. While this is a problem in any architecture, it is more difficult to solve in P2P systems because there is no authoritative source that can withhold information from a player based on what they can or cannot see. Even with distributed authority, peers can misuse this in ways they could not have under an authoritative server, and it requires additional systems for checking.

Gauthierdickey et al. describe protocol-level cheats in overarching categories, but these boil down to spoofing, modifying, suppressing, delaying, or forwarding packets. For example, creating artificial lag by using a "lag switch" or "tapping" to delays packets, which can for example allow players to slow down their position updates in order to teleport or fast-forward from the other players' perspectives. This can be done with a physical device, or through software that fills a lot of bandwidth at opportune moments. There are software and hardware approaches to combating this [127], and games might disconnect you if you lag too much, however sometimes the lag is legitimate. More

advanced methods use firewall or router rules for traffic shaping and a cheater will adjust limits on both bandwidth and latency to stay relevant to a P2P network while still having a considerable advantage over other players.

Along the same line, packets may be delayed for “look-ahead” cheating, where a cheater will wait to see what other players do first before committing to their own action. This can be mitigated through different synchronous and asynchronous P2P lockstep protocols at the cost of latency.

Cheaters can also disconnect right before they are about to lose such that the loss is not registered. More elaborate methods of cheating are modifying the sent updates (e.g. by modifying timestamps), sending different updates to different players, or colluding with players by forwarding them packets that are outside of their AOI.

There is a lot of notable general-purpose, anti-cheat software that in many cases will scan the client device’s memory to determine if they are running known cheat programs, in addition to trying to detect third-party or unsigned code. This software has seen backlash by privacy advocates and in cases where players were banned based on false positives. Valve’s Anti-Cheat would also scan players’ hard disk drives, and was rumoured to receive browsing history [108].

Other anti-cheat measures may include client obfuscation, which only slows modifications down but does not prevent it entirely, or human moderation / spectators policing a game. Even when players are banned by account, IP, or other means, it is possible to circumvent that, so cheating will always be an ongoing battle.

All that being said, fundamentally, the difference in P2P systems is that there is no authoritative server that can validate state updates or behaviour, so the peers must do so themselves. This can create a lot of client-side complexity depending on the game.

This is not limited to VEs either. Any P2P that distributes authority and may rely on voting or majority consensus, which opens these system up to Sybil attacks, where individuals try to gain control of a network by creating many fake identities [35]. This is often mitigated by making it more difficult to create these identities, or otherwise making it prohibitively expensive to gain control (e.g. proof of work or proof of stake in blockchain networks).

DHTs all suffer from this problem if a bad actor manages to spread a lot of fake nodes throughout the keyspace. Some DHTs try to mitigate this through blacklists, but it is an open problem. It is also possible to compromise a DHT with fewer nodes, provided they are all clustered near each other in the keyspace. This is harder to do as it requires bruteforcing to compute the required hashes. I2P’s implementation of Kademlia tries to

mitigate this by appending a date timestamp to the hashes used in the distance function[3]. Thus the keyspace is “rotated” daily, meaning an attacker would have to bruteforce new hashes daily in order to have a set that are close together.

Systems for making a P2P network more resilient to these kinds of attacks do exist [19, 120] including ones that utilise social network information to do so [92, 95, 109, 30, 24]. This leads us to the idea that homogenising centrality within a network such that no single node is too central is likewise a mitigation. This is something we incorporate into our solutions discussed in chapter 4.

2.5.3 Other NVE challenges

Avatars are not the only thing that need to be synchronised in a VE. Other objects, especially those that need to physically interact with others in a deterministic way, need to be as well. Physical objects, subject to the simulation of the physics engine, need to have a consistent view for all players. This holds true for P2P as well as client-server VEs.

Often, a single peer is made responsible for an object, simulates any physics or behaviour for it on their end authoritatively (although other peers can do the same and synchronise with the authoritative peer for that object periodically). Simple Object Management (OM) schemes will have the peer that spawned an object manage it, however some objects may not necessarily have an “owner” and they might persist after the user has moved away, entering the AOIs of other users. All of this is in the scope of OM and has been researched thoroughly throughout the past two decades, much more so than Update Dissemination (UD).

On the topic of persistence, a question that is often tackled is how to handle persistence of game objects and changes to the environment. There have been extensive surveys on the work in this space [49]. In a client-server architecture, the server, which must necessarily always be active in order for the game to be playable, can simply store persistent game objects in a database. If the game comes preloaded with common game assets, this can be as simple as storing object IDs and locations, or deltas from the initial state.

In a P2P architecture however, the peers must store this information somehow. It is not enough to have the peers responsible for a change store the information relating to that change, as once they are offline, nobody can know about it or query that data. Nor

[3] `https://geti2p.net/en/docs/how/network-database#threat`

is it enough to have peers synchronise their local database of game object states (not to mention that this does not scale at all). For example, if peer A made changes that peer B saw, then peer A went offline, peer C can still see those changes as long as B is still around. However, if both A and B go offline, and C makes changes to the same area/data later, we have a conflict.

There are some workarounds to avoiding problems like these, however these are prescriptive to game design. For example, in the nature of these changes and how they are stored (e.g. Conflict-free Replicated Data Types (CRDTs)[119]). Alternatively, game-layer constraints can be used, for example an object/area "ownership" mechanic that dictates that the peer that is the owner needs to be online for an area to be editable or even enterable, and making them the authoritative source on the state of the persistent objects they manage.

Nonetheless, the problem of persistence is tangential to our focus, and can generally be solved entirely separately through various P2P distributed storage systems optimised for replication and search, such as IPFS [5], CFS (which uses Chord and is connected to the same group that developed it) [29], or even BitTorrent in theory, especially since magnet links / infohashes (in conjunction with using a DHT for tracking seeders) allow content-based addressing of stored data.

2.5.4 NVE avatar motion mobility traces and their uses

Mobility traces in the context of NVEs are logs of avatar motion over time. We use these later to both drive the design of our P2P system, as well as evaluate it. Research on mobility in VEs is rich and varied. These are often used in evaluating network architectures whose performance are affected by mobility. The most common type of evaluation workloads are synthetic ones. Simply having avatars move in random directions is however too simplistic. The best approaches try to simulate realistic mobility as much as possible.

A myriad of mobility models have been published with some more well-known than others. Itzel et al. attempt to classify some of theses based on a set of characteristics [67]. Most of these are based on mobility traces from the social MMOG Second Life [91, 93, 88] with some based on the Massively Multiplayer Online Role-Playing Game (MMORPG) World of Warcraft [142] and in one case the FPS Quake II [126]. There are also tools for generating mobility data from a laundry list of these [1].

The most prevalent mobility model in the literature by far is the Random Waypoint Model (RWP) which originates from the Mobile Ad hoc Network (MANET) space [71]. In it, nodes pick a random waypoint, move towards it at a random (within a set range)

constant speed, and pause for a random amount of time before repeating the process. There have been improvements over this model that make it more realistic, which we discuss in this section, however this model on its own is more realistic that a simple random walks, or random direction model. Further, it is argued that artificial mobility models like these will never be as realistic as models based on real traces [67] and that using real traces on their own is also not ideal, as they offer less control over various parameters and may not generalise to other VE types and game genres.

One such parameter is the number of avatars in an area. We note that an analysis of data collected from Second Life by Varvello et al. [133] showed that the distribution of players across areas if very skewed. 30% of Second Life Regions are consistently empty, 45% consistently have 5 or less avatars, and only 5% of Regions reach 30 concurrent avatars as a maximum. It is not uncommon that players concentrate around hotspots or popular areas in social MMOGs, in which case traces from just these popular areas are more than sufficient for evaluating P2P models.

One prominent mobility model is Blue Banana [91] that treats players as a state machine with three states: halted, exploring, and traveling. The transition probabilities between these states were derived from Second Life mobility traces. At each state, a player moves differently.

This model was used to generate synthetic data to evaluate [32]. They also generated data based on "action game" mobility (as opposed to "social virtual world") from a soccer/football game dataset, to cover different scenarios.

The main issue with mobility models such as these is that they ignore higher-level behaviour, such as avatars moving in unison, or clustering in hotspots in a social setting. For games such as FPSs, players on the same team may cluster while avoiding enemies. Often this kind of nuance will affect the ultimate ideal P2P overlay quite significantly.

Other [9] generate data from traces based on a specific game, such as Quake III, that adhere very closely to real game traces [10]. To do this, they go one step further and have up to 32 bots interacting on maps and performing pathfinding over waypoints connected in a Voronoi diagram (this is very typical for FPS AI). They then follow probabilistic models for entering fights, staying at waypoints, or leaving waypoints. This is of course difficult to generalise to other game types, let alone games of the same type, but gives realistic results.

pSense [118] strikes a balance between these two extremes as they simulate player motion based on two models. The first is basic random motion, where players move in a random direction with a probability that this direction can change each simulation step.

The second introduces "hotspots" where players perform random motion around hotspots and randomly move from one hotspot to another. The first spreads players out uniformly across the world, while the latter clusters players around these hotspots and the paths between them, which is closer to reality in many cases.

The advantage with mobility models is that they allow us to scale up the number of simulated players to stress-test our systems at the cost of realism. They should be used in conjunction with real mobility traces wherever possible in order to not fall into the trap of oversimplified simulations. Research on mobility in virtual environments is rich and varied.

2.6 The underlying network

2.6.1 Application-layer multicast

It is important to note that our discourse is solely with regards to Application-Layer Multicast (ALM). Multicast (or indeed AOI-cast) is abstract at the level of the nodes in the various overlay networks, and the traversal through the underlying physical networks is unknown, manifesting itself indirectly only when we take node-to-node network measurements into account. That is what makes an overlay network by definition.

We draw the distinction between this and IP multicast (note that UDP can use multicast addressing) other multicasting at lower layers. Some previous work relies on this kind of multicast, especially in the area of MANETs, where the smartphones are connected in a Wi-Fi Direct or Bluetooth mesh. This will likely gain momentum as location-based Augmented Reality (AR) games become more popular, but is for the most part outside of our scope, except that there is some overlap in the literature with simulations for evaluating these systems (e.g. human mobility can simulated in similar ways to avatar mobility with the Random Waypoint Model etc, which we discuss in §2.5.4).

Within the ALM context, we also draw the distinction between multicast groups based on P2P connections (e.g. P2P "rooms" in Diablo) and multicast groups further abstracted (e.g. Internet Relay Chat (IRC) channels — while users within a channel chat only with each other, all traffic goes through the IRC server). ALM in the context of our work refers to the former.

2.6.2 Network coordinate systems

In the design of internet applications, it is immeasurably useful to be able to model and predict point-to-point network conditions. These models are needed as it is usually impractical to continuously have every node in a network ping every other node in a network to understand these network conditions. Oftentimes it is not even possible to do so, or a latency measurement is a precursor to routing, and making that measurement itself a significant overhead.

Network Coordinate Systems (NCSs) seek to model networks in a way that is useful for prediction and other purposes. These usually focus on latency between nodes. Among NCSs, Vivaldi [28] is indubitably the most widely used, primarily because it is very easy to implement, can be distributed, and gives good results. It models the network as a spring system, where the latencies correspond to a spring's resting length. This idea is often used in data visualisation when rendering force-directed graphs — Hooke's law (which describes the behaviour of springs) is applied to edges and Coulomb's law (which describes the behaviour of charged particles) is applied to nodes.

Other NCSs solve some of the limitations of Vivaldi at the cost of complexity. For example, Pharos [22] has better accuracy than Vivaldi by using two separate overlays with two separate sets of coordinates for long and short link predictions. Meanwhile, Phoenix [23] solves triangle inequality violations by using a matrix factorisation model.

Other NCSs include Global Network Positioning [106], which instead uses "landmarks" to allowing hosts to triangulate their position by pinging at least three, among many others that have been extensively surveyed in the past [34]. There has also been a lot of surrounding research, for example on the dimensionality analysis of these coordinate systems. Vivaldi empirically demonstrates that going from two to three dimensions gives only marginal improvements, and even less beyond three, likely due to the fact that latencies can be strongly correlated with physical distance, and the physical internet is mostly two to three dimensional. Others have studied the effects of spikes and outliers on the stability of an NCS, and how they can be made more stable by essentially putting latency measurements through a low-pass filter [44].

There are however other methods that do not rely on continuous sampling, such as MIT's King [55], which piggybacks off of DNS infrastructure, and iPlane [94] with predictions based on measurements from known vantage points. The author of iPlane has also released a log of these measurements, which we use to model our networks in chapter 4.

2.6.3 Network level of detail

Many researchers have had the idea of attenuating the frequency or granularity of updates for peers that are further or less important, instead of simply communicating with all AOI neighbours equally. For example, Kawahara et al. split peers into “active” and “latent” with the latter being updated at a lower frequency [77]. Similar schemes to this followed [20]. On a similar vein, Kenny et al. dynamically adjust transmission rate based on network conditions [83]. Fibocast adjusts the update rate of a peer dynamically based on a Fibonacci sequence and the distance (in hops) another peer is from them [70]. This can have a tangible effect on topology evaluation metrics.

We coin the term Network Level of Detail (NLOD) based on the analogous concept in computer graphics, Level of Detail (LOD), where 2D sprites / 3D models may be rendered at different resolutions or complexities respectively based on how far away they are from the camera. We touched on this briefly earlier when discussing quadtrees/octrees, as these are ideal data structures for storing LOD data in.

There are other rendering optimisations, such as not rendering occluded geometry, or geometry that is outside of the player’s field of view, and there are indeed network equivalents to this [7]. It makes sense after all that a player would not need any (or fewer) position updates from another player who is behind them and they cannot see. The challenge here is to communicate changes in a player’s field of view in time to receive these updates.

Some NLOD methods even consider social proximity in addition to physical (virtual) proximity [56]. Visual information sent to a player can range from low definition to very high definition based on this combined metric. The intuition behind this is that players care more about high-fidelity data from players they are socially close to.

NLOD techniques are not as widely researched as general P2P topologies, and are usually included as an afterthought. Part of the problem is that these techniques are prescriptive to game design and can be difficult to generalise. This is an area we explore further in chapter 4.

2.7 Overlay networks

So far, we have discussed background information that frames the context of our work. As outlined in §1.3, one of the primary objectives of our work is to design a P2P overlay network topology optimising for specific use cases and deployment environments. In this

section, we discuss related work then critique where it falls short, to motivate the need for further work. We also discuss neighbour selection metrics as well as routing in a P2P context, to better paint a picture of where the challenges and our focus lie.

2.7.1 A taxonomy of existing topologies

Existing work has been categorised into *structured* and *unstructured* P2P [137, 49]. Generally, structured approaches utilise DHTs or similar distributed data structures, while unstructured approaches connect peers based on factors such as VE positions. We focus on unstructured systems as our contributions lie in using factors such as these to build more optimal topologies.

To elaborate, literature focuses on P2P architectures for different purposes. One common purpose is OM, where the question is which peer should manage a set of virtual objects, or which peer should be responsible for a particular region. For purposes like this, it is trivial to assign peers and objects IDs and map them to a DHT. Derivatives of this that take account of object and avatar locality perform better. These still use DHTs, but with locality-preserving hashes, usually with range-query support [17, 131].

These architectures also require a means of UD. To do so efficiently, previous work explores different methods of neighbour selection based on semantic distance between peers. Other methods look at network distance between peers, but in any case, updates propagate through a sparser network than the naive completely connected variant. Given a set of peers and information on these peers, we aim to design an algorithm for building a peer network topology that optimises for the metrics discussed in §5.2.5.

A myriad of approaches to the different facets of distributing VEs exist in the literature. UD, which we focus on in this work, is much more niche. Usually the more well-known systems focus on OM, and pick obvious/default techniques for handling the UD side. Survey papers on P2P systems for VEs generally ignore this side too because it is so rarely explored. UD itself is not a term that is as widely used as IM and OM and the method is usually indirectly implied.

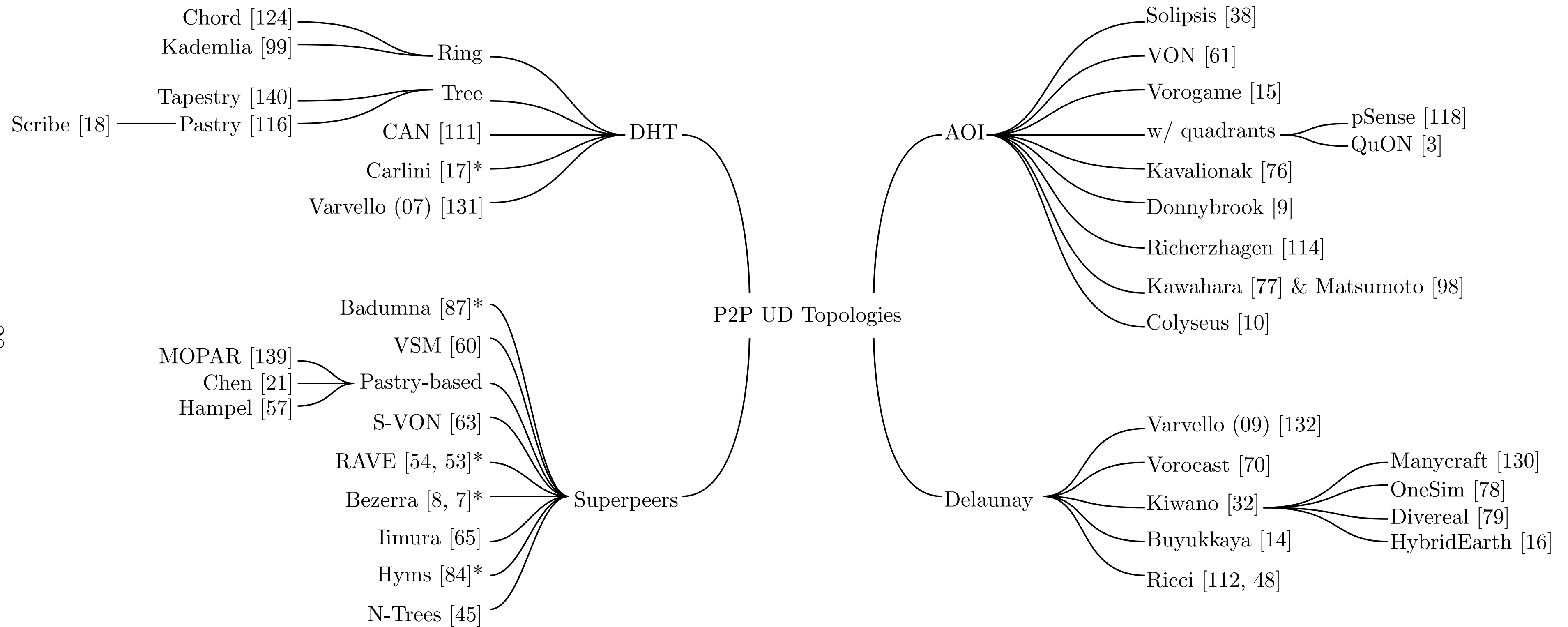


Figure 2.2: A taxonomy of update dissemination topologies of major P2P architectures
∗ — hybrid P2P with dedicated servers

Figure 2.2 presents a taxonomy of the UD topologies. This includes the major architectures/protocols in this area, but also some lesser-known work. These can generally be split into four high-level categories according to what the method in question is based on. These categories are DHT, AOI, Superpeers, and Delaunay.

For methods that have been named, we refer to these by their name. For those that have not, we refer to them by the surname of the first author of the associated publication. Citations to these publications are linked in figure 2.2 — throughout this chapter we will refer to them by their names in figure 2.2, as opposed to citation number.

We also note that, even though these publications range across the past two decades (with a couple even earlier), the specificity of this area has the consequence that it is dominated by around a dozen research groups. We therefore raise this caveat as an additional point when considering the categorisation, as some methods are by the same authors or authors associated with them in some way.

We now examine these four categories and any points of discordance therein.

2.7.1.1 DHT-based

DHT-based P2P architectures are very popular and hardly limited to our research context. Their power becomes evident especially in distributed storage and search of data over n nodes, as this data can be found in $O(\log n)$ hops. Freenet[4], its successor Gnutella[5], BitTorrent[6] (for distributed tracking) all make use of DHTs.

This class of topologies is what VE literature will classify as *structured* P2P, and while there is a strong prevalence of / preference for *unstructured* approaches, it is still important to consider these. The "big four" publications in this space were Chord [124], Pastry [116], Tapestry [140], and CAN [111].

Chord's architecture is what most people think of when imagining DHTs. Node IDs and keys are hashed to create identifiers such that they map to the same space and are arranged in a ring. Each node is responsible for the storing the values for the keys closest to it. Each node also maintains routing information for the nodes 2^i hops away from it, for values of i from 0 until where 2^i wraps around the ring, in a "finger table". Thus, any node can reach any other node in on average $O(\log n)$ hops with n nodes.

Practically, Kademlia [99] has also seen expansive use. It is similar to Chord, except the distance between two nodes with identifiers a and b respectively is $a \oplus b$, as opposed

[4]https://freenetproject.org
[5]https://web.archive.org/web/20080525005017/http://www.gnutella.com/
[6]http://www.bittorrent.org/

to Chord's $(b - a) \mod 2^n$. This has the useful mathematical characteristic of being commutative (the distance between nodes a and b is the same as the distance between b and a) which means a node is more likely to receive a lookup message from one in its routing table. Kademlia exploits this by coupling routing table updates to lookups, which improves overall efficiency.

CAN does not use the same ring layout, instead peers are mapped onto an n-dimensional virtual coordinate space. Every time a new peer joins, the position it maps to is split between it and the peer previously responsible for that area. The original paper [111] explains this intuitively in two dimensions (like a simpler form of quadtrees). Beyond this, keys and values are mapped to the same space and distributed among the peers just as with any other DHT-based system.

Pastry and Tapestry are very similar in that they form prefix trees. Each node holds the routing information for peers that have incrementally more prefix digits in common. For identifiers of length l, the first set of peers share the first $l-1$ digits in their identifiers, the second set $l-2$, and so on, up to a set of peers that have identifiers with no common prefix of any length. The result of this is that peers have more connections to other peers with identifiers that start the same way, and that a peer can reach any other in a predictable way by routing down the prefix tree.

The differences between Pastry and Tapestry are minor — the underlying topologies are the same, but Pastry mainly also performs object replication. Scribe is a pub/sub system build on Pastry, which exploits the underlying architecture to create multicast trees through which messages can be propagated to any peer who subscribes to a relevant topic. Scribe's fault tolerance is coupled with Pastry's object replication, since if the root node of a topic goes offline, there are still backup root nodes for that topic. It is a stretch to call Scribe a DHT-based topology however, since it only really uses a DHT (Pastry) to build multicast trees, so if it were used for UD the data would not be pushed through the DHT itself.

Carlini additionally maps Virtual Nodes over a DHT such that peers in a VE can manage any number of these and they can be passed around for load balancing. They also allocate "backup" virtual nodes that are run on authoritative servers to mitigate cheating.

Both Carlini and Varvello (07) rely on locality-preserving hashes for their DHTs, and rely on the range query support of these systems in order to query for peers within a certain AOI. These are especially useful for the countless number of instances where DHTs were used for OM, as having objects map closest to the peers that are closer to them in the virtual space is an embarrassingly obvious use of this.

2.7.1.2 AOI-based

As mentioned before, there is a clearly exploitable characteristic of VEs: the locality of peers in the virtual space correlates strongly with the communication requirements of a peer at any given point in time. More specifically, peers are interested in receiving updates from nodes within their area of interest. For most systems, it therefore makes more sense that a node communicates directly with nodes within its AOI, as opposed to, for example, its n nearest neighbours.

Before we explore the various AOI-based methods, it is important to note that many of these are prescriptive — they either imply that the application behaves a certain way, or they require it. The most common instance of this is in prescribing how AOIs are defined. This is difficult to generalise (sometimes making a method limited to a type of — or even a single — game). We maintain that generalisable UD mechanisms should not be prescriptive of game design, rather the opposite should be true.

We therefore reduce AOI-based topologies to the same type even if they introduce different ways of building dynamic AOIs. Similarly, AOIs that are based on rectangular/hexagonal/quadtree/Voronoi cells, rather than circles/spheres with a set radius, all count as AOI-based. These different IM schemes have been surveyed extensively [13].

Solipsis [80, 38] and related work by the same researchers completely connect peers within an AOI. Kavalionak, Colyseus, and Donnybrook do the same, as they focus more on other aspects of the architecture, such as OM. VON, one of the most well-know P2P systems for VEs, as well as other previous work by the same authors [62], make some improvements by using dynamic AOIs with connection limits, but fundamentally peers are connected within an AOIs. The same holds true for Vorogame.

pSense peers additionally connect to four nodes outside of their AOI in each quadrant, but the primary purpose of these sensor nodes is to detect new incoming nodes as the peer's avatar moves, and is not related to UD. Some networks, such as Richerzhagen, are AOI-based, but explore different dissemination protocols, such as different kinds of flooding (e.g. probabilistic flooding) even though the overlay network topologies are the same. We explore this further in later chapters.

2.7.1.3 Superpeer-based

Unlike dedicated servers in a datacenter, peers can exhibit a lot of heterogeneity in available resources and bandwidth. Superficially, it makes sense that more powerful peers should have a higher centrality in a P2P network, and handle a greater share of

forwarding or management. Indeed, many famous P2P applications, such as KaZaA or pre-Microsoft Skype use superpeers, even if they might call them by different names. When it comes to performance, and in some cases fault-tolerance and integrity, literature has shown that a superpeer architecture is superior to a pure P2P one [138].

Architectures like S-VON select superpeers based on capacity, stability, and trustworthiness. S-VON also uses an NCS for selecting relays. Similarly, HYMS selects superpeers to manage spatial cells based on the CPU, memory, and bandwidth of a peer. HYMS additionally incorporates replication on authoritative servers, which is common with these hybrid approaches to address bad actors in the network and cheating. Other systems try peer specialisations based on their resources (memory peers, visualisation peers, and connectivity peers) [20].

These superpeer architectures result in tree structures, usually coupled with some sort of zoning (rectangular, hexagonal, etc). Where these methods differ is how they select they zone, and how they select a superpeer responsible for a zone. Often, DHTs are used for superpeer selection, for example in Iimura "Zoned Federation" with rectangular zones, Badumna with dynamic AOIs, and VSM with Voronoi cells. DHTs with a good hashing function are a simple way to ensure that the load is evenly spread. As a matter of fact, MOPAR, Chen, and Hampel each use Pastry as a DHT with hexagonal zoning. Hampel additionally takes into account peer resources.

Other methods, such as N-Trees, are similar to CAN, and couple superpeer selection and zoning much more tightly. N-Trees are like quadtrees (2D) or octrees (3D) but for N dimensions. Once an N-dimensional cell exceeds a certain capacity, it is split into 2^N cells, and so on. This is a powerful tool in computer graphics when e.g. rendering a heightmap — one can have different levels corresponding to each node in a tree, such that a node is an average of all of its children. Then, different parts of the heightmap can be rendered with different levels of detail, based on distance from the camera. This can likewise be applied to a superpeer network, where when a cell reaches capacity, a peer becomes a parent to the other peers in that cell and it is split.

Many superpeer architectures are not fully P2P and make use of authoritative servers for replication, cheating mitigation, reliability, and other purposes, not to mention bootstrapping. These hybrid systems are marked with asterisks in figure 2.2. In fact RAVE and Bezzerra describe techniques for distribution of clients across servers, and are not P2P at all, but were included as these techniques are analogous to distributing peers across superpeers. There are many general hybrid approaches that rely on centralised servers and simply use P2P networks to offload some traffic [69], however these are largely out of the scope of this thesis.

2.7.1.4 Delaunay triangulation-based

The fourth type of prevalent UD topologies are those based on Delaunay triangulation; an old and very well known triangulation algorithm. The graph created by Delaunay triangulation is the dual graph of that created by Voronoi tessellation. It is easy to visualise this intuitively in two dimensions, as the "cells" in a Voronoi diagram of a set of points are simply the area within which any additional points would be closest (for some definition of "close" — usually Euclidean distance) to the Voronoi point of that cell. Delaunay Triangulation, being the dual of a Voronoi diagram, therefore results in even triangles, avoiding slivers, unlike other tessellation methods, which is why it is especially useful for graphical use cases. An important point to note, which we will revisit later with our own approach, is that a Euclidean minimum spanning tree is a subgraph of a Delaunay triangulation.

Work like Varvello (09) connects nodes in a straightforward Delaunay network, however other work adds to this. For example, Kiwano, which has been practically applied several times, instead found that the 3rd power graph of a standard Delaunay graph provides a better performance trade-off. Buyukkaya (which uses Voronoi cells for OM) extends Delaunay triangulation to avoid the rapid "flipping" of edges when a peer is right at the threshold of a location where that would happen if they moved back and forth.

Vorocast additionally creates a spanning tree within an AOI and makes sure that updates disseminated down these trees do not reach the same child twice, as might happen with general flooding. While this may reduce traffic, we argue that it detracts from the advantages of having redundant paths, including reliability and validation.

Ricci (work from the same group as Carlini previously) uses yet another routing heuristic within the same topology: compass routing, which has been used in the past in creating spanning trees for ALM [2]. While this can in theory cut down on latency, it can also raise the worst case, in situations where a packet goes down a dead end for example.

2.7.2 Neighbour selection metrics

So far, most of the systems we have discussed incorporate some notion of semantic distance between peers and peers will select or be assigned neighbours based on some selection scheme that takes these distances into account. This is ordinarily the virtual position between the avatars of said peers, although some architectures may take network distance into account in some shape or form. Considering network distance is more prevalent in

superpeer selection or similar, such as the use of NCSs for relay selection in S-VON [63]. We discuss NCSs in more detail in §2.6.

Sometimes geographic distance is used instead, either as an approximation of network distance, or because geographic distance and virtual distance literally overlap (think location-based AR games like Ingress[7] or Pokémon Go[8]).

Superpeer selection based on the resources of peers implicitly affects the topology of a network. While these selection schemes cannot be reduced to a selection "metric" per se, peer resources can be a factor in neighbour selection, giving peers more "gravity" in spatial schemes, or a higher likelihood of being connected to in probabilistic schemes.

There is no reason why other metrics cannot or should not be used in spatial schemes however. For example, a recent survey on socially-aware P2P systems has shows how social metrics can augment and improve search efficiency due to homophily, as well as other benefits (including trust-based defences against malicious peers).

Elaborate trust and reputation management systems exist for P2P VEs [64] that can also be incorporated into semantic distance measures, providing protection against malicious behaviour for free at the topological level. We explore this idea in more detail in chapter 4.

Other work [56] has achieved performance improvements through limiting the avatar details transmitted from peer to peer based on their social proximity, from the assumption that players are less interested in the appearance of avatars they are not friends with in a social VE. The benefits of incorporating these kinds of metrics can therefore have performance benefits in addition to combating malicious behaviour.

2.7.3 Routing over P2P NVEs

In many cases, topology and routing are strongly coupled. For example, once DHT routing tables have been constructed, the obvious routing scheme is to send packets to the peer with the hash closest to the target. There is no real overlay network per se since pairwise links are connectionless.

This might not always be so straightforward however. After a P2P topology has been constructed, there can be multiple possible routes from a source to a target, and the optimal route can change over time depending on many different conditions.

We have already touched on methods that route messages differently within an AOI

[7] https://en.wikipedia.org/wiki/Ingress_(video_game)
[8] https://en.wikipedia.org/wiki/Pok%C3%A9mon_Go

beyond simple shortest-path routing or flooding approaches — recall the Delaunay-based topologies such as Vorocast, which creates spanning trees with in an AOI and avoids children getting the same update twice via different routes, or compass routing in Ricci. The list of different routing protocols is endless, many of which take locality into account, such as greedy forwarding, face routing, and other geographic routing protocols. These protocols have been extensively surveyed in the past [125]. UD is however, by definition, not single-path, so many of these strategies are not applicable, and flooding must instead be used, which offers far less flexibility. Fundamentally, this is more of a load balancing problem (think backpressure routing) than it is a routing problem. This is also why we believe that it is doubly important to consider network conditions, and not just virtual positions, when constructing P2P topologies. We revisit this in §4.4 in our design chapter.

Generally speaking, different schemes all try to optimise for metrics such as consistency (we discuss this in more detail in the next section) which ultimately depends on the latency between peers, but most systems only do this indirectly with their neighbour selection methods. Even knowing latencies between peers is not enough, since naive shortest-path routing will naturally lead to traffic concentrating at nodes with higher capacities, leading to congestion, which will in turn lower performance. Because it takes time to query latencies, all measurements are necessarily stale, which leads to the flocking behaviour oscillating. This is however a well-known problem with many solutions [113].

The main difference between load balancing across servers versus peers, is the inherent heterogeneity of peers. Some rudimentary structured P2P load balancing schemes attempt to achieve better performance by classifying nodes into "heavy" and "light" peers [110]. This is too far removed from the fundamental goal of dynamically minimising latency, and it can be argued that looking at latency alone will indirectly signal other factors such as congestion, which we explore.

An additional difficulty comes from the fact that there can potentially be secondary goals, such as redundancy through sending the same message via multiple paths simultaneously (as with flooding) to combat packet loss or malicious peers. Routing such as in Vorocast falls incredibly short when it comes to meeting these kinds of goals. This is also a consideration we take into account when designing our solutions.

2.8 Evaluation metrics

Previous work has sought to evaluate the quality of P2P solutions in various ways. Unfortunately, this area (even at the level of general-purpose P2P) does not have as much

research activity as, say, computer vision and/or machine learning research with well established benchmarks and datasets for evaluation. That being said, there is a general pattern to how these systems are evaluated. These usually boil down to a series of quality-of-service indicators. Here we briefly discuss the most significant ones.

2.8.1 Consistency / Staleness

Schiele et al. [117] laid out the requirements of P2P MMOGs and *consistency* was second only to *distribution*. Consistency is probably the measure that is most prevalent in the literature in one form or another. The oldest instance of this that we could find was a definition of consistency from 1999 [33] where an absolute consistency requires that all peers see the exact same information at the same time. This is impossible since there will always be network latency and we are limited by the speed of light, but any divergence from this ideal is defined as *inconsistency*.

Existing work [132] also measures quality of experience via inconsistency. However, they define this as the fraction of missing/incorrect information within an avatar's AOI, for example avatars having outdated coordinates. In other words, whether or not a peer is inconsistent is binary, and the full metric is a ratio of inconsistent to consistent. They also measure *inconsistency duration*; how long an avatar's state is inconsistent.

Other solutions [9] set a time threshold, for example 150ms (in practice this is dependent on the kind of game) and any updates that pass this threshold are considered stale. The quality of a network depends on having a low proportion of stale updates. Itzel et al. [66] call this *inconsistency tolerance*.

Richerzhagen define staleness on a per-peer basis — the time elapsed since an update has been received from a peer. This is plotted against AOI radius. This metric has limited utility however as it combines UD delay as well as the general update frequency for any given node. It might not necessarily affect user experience so drastically as most games will incorporate some level of position interpolation/prediction.

In work by Kawahara and Matsumoto [77, 98], as well as VON, "consistency" instead refers to topology consistency. It is defined as the ratio of correctly known AOI neighbours per peer (with the global consistency at any given point in time being the mean of peer consistency). In other words, if a peer has a neighbour list of a, b, c, d when it should be a, b, c, e, that is a consistency of 0.75. They plot this against the velocity of avatars for various AOI radii. This is less useful as a metric for comparison of different UD topologies (especially where there is a server with global knowledge coordinating) because it will naturally be very similar for all topologies. It is more useful for determining ideal AOI

radii with respect to in-game player velocities for a system.

Finally, to confuse terms more, other significant work [70] define the subtly different "neighborship consistency" as the ratio of known AOI neighbours to actual AOI neighbours. Systems like VON, which send updates to all AOI neighbours directly, will always have a neighborship consistency of 1.0. However, Vorocast, Fibocast, and any topology that involves peers within an AOI forwarding to each other, can have a lower neighborship consistency. In [70] this is plotted against the number of nodes in the network, however it is conceivable that this can be plotted against different AOI radii or shapes. This is again limited in utility for the same reasons as above.

Metrics such as these rely on having "ground truth" data for the timing of avatar position updates and are therefore best suited to simulations from traces or synthetic data where one has global knowledge.

2.8.2 Bandwidth

It is not uncommon for P2P network evaluations to consider bandwidth and/or individual peer upload/download volume and rates [118, 9]. Naive P2P approaches have peer bandwidth requirements that scale quadratically, therefore it is important that well-designed approaches mitigate this. As such this is one of the more obvious metrics to consider.

A model of peer upload capacity has been built from BitTorrent and speed test datasets [9]. Models like these become outdated quickly however as bandwidth grows worldwide. It is much more important to consider how traffic scales with the number of players, and how it is distributed across players.

pSense bandwidth evaluations select 100 random nodes and traffic (in and out) per node (max and average) over time. More importantly, they plot per-player bandwidth against average players in vision range. Results like these are important as some topologies (e.g. with superpeers) inherently put more load on certain peers, which can in turn create congestion at specific nodes. This is especially critical as individual peers have much more limited available bandwidth (especially upload) when compared to dedicated game servers in a datacenter. A simulation where peer bandwidth capacity can simply increase with the network size indefinitely is not realistic.

Vorocast similarly considers bandwidth per number of peers in the network, but like many other systems, does not take packet loss, latency, or other physical network characteristics into account.

2.8.3 Upload/download per node

On a similar vein to bandwidth usage and capacity requirements, examining the upload and download of nodes in a P2P network is another obvious and common evaluation method. It is most useful to look at distributions of these against the number of nodes in a network, however some work has used this to justify parameters of their systems, for example AOI radius, by investigating how upload/download change for different AOI radii [114].

Crucially, serious evaluations will avoid basing quality judgements on averages of these, as these distributions can include far outliers across nodes (such as in superpeer networks where some nodes may handle much more traffic relatively) and even spikes in upload/download for the same node. A such, these are generally plotted as box plots [114], or in the case of VON (a system with dynamic AOIs), separately for the average and the maximum [61].

2.8.4 Protocol Quality

pSense [118], considered a typical example of such solutions [137], describe a metrics they call "Protocol Quality" (PQ). They split the vision range around an avatar into two parts; a small radius for direct interactions (e.g. talking or fighting), and the remaining vision range for less important interactions. Anything outside the vision range has no effect on PQ.

Within the smaller interaction range, the PQ between two avatars is the "Position Age" — the time between a when a position update has been sent by one avatar and received by another. As *pSense* evaluations do not consider inter-peer latencies and other factors, the unit of this metric is simulation rounds. In our vocabulary, this is equivalent to number of hops.

Outside of the interaction range, they take the Euclidean distance between both avatars and normalise it such that it maps linearly to the range between interaction radius and vision radius. In other words, 1 at the interaction radius, and 0 at the (larger) vision radius. Then they raise Position Age to the power of that value. This has the effect of reducing the importance of Position Age with distance, as they argue it is less important for peers to have fresh information as you move further away. Any peers outside the vision radius have no effect on the PQ metric whatsoever.

PQ is first described as a pairwise metric. To measure the PQ for an individual peer is simply an average of PQs for every other peer they are connected to. Those averages

averaged yet again are the PQ for a simulation round, and finally all rounds in a simulation run are averaged again to create the overall PQ for the whole solution/simulation run.

There are many flaws with this method of evaluation. First the fundamental assumptions about game mechanics. Not all games have a notion of distance-based levels of importance/detail, and an interaction at a distance might very well be as important as one close up. For example, in an FPS, shooting an opponent through a sniper scope is a critical interaction at a distance within a vision range.

Further, the actual radii that *pSense* used were not discussed, nor was the justification behind picking reasonable radii. This assumes it is even possible to pick general-purpose radii, since different types of games have different parameters (cf the FPS example).

The units are also very rudimentary, since they focus on hops rather than latency, which assumes that inter-peer latencies are uniform. In practice, this is not the case, and the heterogeneity in latencies is too significant to ignore.

Aside from the formulae for calculating PQ being rather arbitrary and only superficially justified, valuable information is lost by all the averaging. It is just as important to consider distributions and upper bounds, as a few outliers with terrible PQs will be ignored by this metric.

2.8.5 Delay

A straightforward metrics that is not limited to NVEs, is the delay/latency between pairs of peers [114, 141]. This is usually plotted against the number of nodes to evaluate scalability.

Work by Rooney et al. [115], while tangential (multicast reflectors within static AOI subscriptions for P2P games), build other abstractions on top of this, such as the concept of “TooLate packets” (the frequency of which increases as they artificially increase packet loss ratio). Metrics like these require a latency threshold, which practically speaking is game-dependent (see table 2.1), beyond which incoming packets are too late.

Similarly, they define “RelevancyTime”, which is the time beyond which a packet is no longer relevant and should not be retransmitted. Incorporating retransmission into a system in this manner can improve other metrics, without incurring the disadvantages of full TCP. This is similar to what Itzel et al. [66] call *interactivity*, except it is based on the action (i.e. some kinds of state updates can have much tighter latency tolerances / interactivity requirements).

They also introduce *priority*, which would allow consistency management schemes to

selectively priorities state updates based on type. For example, in the case of congestion resulting in consistency degradation, the consistency of update types with lower priorities (such as certain movement updates) can be relaxed before that of those with high consistency requirements (such as virtual trades).

In a VE context, delay is implicitly considered in consistency and staleness metrics, and is therefore less informative on User Experience (UX) than those.

2.8.6 Reliability

Reliability is less explicitly defined, and can ambiguous, however VON uses this in part to mean how consistency (in the topological sense) changes as packet loss (global, artificial (in a simulated environment)) increases, as well as how many recovery steps are needed to bring consistency back to 1.0.

Richerzhagen similarly plot drop ratio against AOI radius [114]. As expected, bigger AOIs result in more collisions and the drop ratio goes up.

In general, when simulating these systems, it is useful to evaluate the effect of packet loss on the various Quality of Service (QoS) indicators.

2.8.7 Drift distance

Drift distance is a metric that is related to consistency, and can be more meaningful in genres such as FPSs, where a player's position drifting from the ground truth position can result in shots being missed and a quick deterioration of player satisfaction, as even near misses can be very frustrating. This metric is in fact very useful when testing position interpolation/prediction schemes.

This metric was likely first defined in 1999 [33] as the difference between a player's position on two hosts. If the drift distance is 0, the game is perfectly consistent. The same work also informally analyses player satisfaction in an attempt to narrow down a threshold for "acceptable" drift distances.

Later, this metric was used in VON and other P2P VE networks [61, 70] and is usually plotted against the number of nodes in a P2P network.

2.9 Evaluation workloads

The second ingredient for the proper evaluation of a P2P solution is an appropriate workload. This should be at the very least avatar mobility data. In this section we characterise the workloads used in previous work and discuss their merits and demerits.

2.9.1 Simulation testbeds

The tools for evaluating P2P VEs are currently still very limited. Most researchers will either build their own, or use general-purpose P2P simulators such as PeerSim [104], which is limited in its utility for this sub-area. For example, PeerSim measures hops between peers, but not much beyond that, which makes it difficult to bridge between those simulations and real networks with heterogeneous links between peers and different load balancing schemes.

Incidentally, Triebel et al. have created a game specifically for evaluating P2P systems [129]. Unfortunately, as we have already mentioned, this can make the scope of an evaluation very narrow if it is only suited to one game.

As part of our work, we have decided to develop and release a flexible evaluation framework for P2P UD topologies, with built-in support for various mobility models and streaming traces for the research community to benchmark their systems under a variety of metrics covering as many areas as possible. We present this framework in chapter 4.

2.9.2 Mobility workloads

2.9.2.1 Characterising traffic

There have been a number of analyses of P2P game network traffic [4, 36, 21]. Sometimes, these are in conjunction with mobility analysis [136] where researchers look at how the two are linked with upload and download traces from individual peers.

Unfortunately, a lot of this kind of research is third-party, meaning the traces are based on data that only individual clients can receive, which is especially limiting in the case of client-server architectures. It is rare that research by game studios with unprecedented access to server-side data is published. For example, Microsoft made great strides in estimating P2P network QoS based on XBOX 360 data they collected from their Halo 3 players [90]. It is impossible for anyone except Microsoft to perform this level of research, as it relied on data that only they have access to, such as the IP

addresses of all their players and when they are active. It would also be difficult to release this data to external researchers as it would violate the privacy of their users. This kind of data will of course always be better than artificial traffic over academic networks such as PlanetLab or Emulab. This is similar to e.g. how the bot/spam-prevention methods of social media giants cannot possibly be replicated by external researchers, as their models do not have the same level of completeness of training data. Since this kind of research is usually motivated by increasing profits, it can be argued that there is also an incentive to keep trade secrets and not publish intellectual property for competitors to see.

Microsoft's results allowed them to predict the quality of network paths (also taking diurnal activity cycles of their players into account). They define "quality" in terms of Round-Trip Time (RTT) and capacity. In the first instance, they use these predictions to decide if they should probe a path. They found that in general, IP addresses are enough to have a good estimate of network paths. If that is not accurate, the estimate can be improved through IP prefix history, and geographic coordinates provide estimates that are accurate enough to filter out low-quality network paths that would definitely have too low an RTT to be worth probing.

2.9.2.2 Latency consideration

When evaluating P2P overlay networks for VEs, related work very rarely considers heterogeneity in the underlying physical networks. This is largely because network conditions do not play a role in the way the overlay networks are structured, and most methods focus solely on avatar virtual positions.

The most a previous evaluation has done in considering the physical network is *Colyseus* [10] which used MIT's well-known King dataset [55] in order to simulate realistic pairwise latencies.

2.9.2.3 The holy grail of workloads

The main problem with evaluations in this area is that generally research activity is limited to academia or third parties, rather than multiplayer game developers and studios. Due to this kind of research being relatively experimental and hard to get right, there is little incentive for game developers to undertake this research with no guarantee that the effort is financially worthwhile. Similarly, there is little incentive to sacrifice the extra storage space to log player locations and other data.

As a result, all workloads in literature (that are either used directly or to build mobility

models) are collected indirectly in a usually painstaking manner, and can have missing data and temporal gaps due to AOI constraints or bots being kicked [93].

The perfect dataset to evaluate P2P overlays for games would include:

- Logs of all avatar position updates
- Pairwise latencies between players; the denser the better
- Traffic logs between players (other traffic beyond position updates, like chat)
- Any other individual and pairwise metadata that may be relevant (e.g. reputation, trust scores, account age, etc)
- Geographic location and/or IP address

A dataset like this would allow systems to be built and experiments to be run at an unprecedented level. A dataset like this could in theory be built passively on the game servers of any client-server multiplayer game.

The second point would require an actual P2P network for players to be able to ping each other, but barring that, there are other existing methods for building an NCS that do not require this, as discussed in §2.6.2. This can be supplemented with geoIP data — this is the only reason we include it in the list above. This also avoids the disadvantage of exposing player IPs to other players.

On the subject of anonymity, mobility data is virtually non-identifying, and any other traffic (such as chat) can simply be represented by a number of bytes. This would allow an accurate simulation of multicast traffic for playback over a custom network. As part of this research, we have collaborated with the developers of an independent, browser-based sandbox MMOG to build a rich dataset that captures as many of these characteristics as possible, so we will revisit this topic again later.

Chapter 3

Characterising Networked Virtual Environments

3.1 Overview

In order to to build Update Dissemination (UD) systems for Networked Virtual Environments (NVEs), we must first understand NVEs themselves. In §1.3 we laid out a set of research objectives with emphasis on designing and implementing a system that can maintain performance under different NVE use cases. While we have already defined and justified what appropriate performance metrics are in §5.2.5, we must still characterise what we mean by "different NVE use cases". This is hardly straightforward.

In this chapter, we focus on capturing NVE constraints and requirements. To do this, we build a large dataset from an existing NVE, and make targeted measurements over this and other collected data. We also run a number of experiments and analyses to understand browser and network requirements.

3.1.1 Measurement objectives

We can begin by making some observations from literature and our knowledge of networked games and other NVEs. We use this as a starting point to make much more explicit, targeted measurements in order to properly set the scene and capture design and evaluation requirements and constraints. Unlike other Peer-to-peer (P2P) systems, NVE variants have some important properties as follows.

Ephemeral connections Optimal topologies are strongly coupled to spatial, application-layer information. Optimal topologies are therefore highly dynamic (moreso for certain genres). This implies that connections are far more ephemeral. For update dissemination in this context, structured topologies (especially those that do not take account of spatial information) will always perform worse than unstructured topologies

Ephemeral peers Peers only remain connected for as long as they are present in an NVE. Depending on the device they connect from and the kind of connection, their connection might also not be as stable. Peer presence is therefore similarly far more ephemeral.

Small, frequent updates over unreliable protocols The disseminated data is small, rapidly changing, and becomes quickly irrelevant. User experience therefore has much stronger links to latency over other network metrics. Retransmission is pointless, making unreliable communication protocols the standard. This data is never stored and has a single authoritative source so it is also not replicated. Generally, the disseminated data (e.g. position updates) can be predicted and interpolated, and therefore application-layer optimisations can improve performance.

Cheating There are incentives for malicious behaviour which can manifest itself in many ways. Some of these are unique to P2P NVEs.

In this chapter, we take an empirical approach to capturing the constraints and requirements of NVEs by collecting and analysing a real NVE that fits our research context exactly. More specifically, a large-scale, browser-based Massively Multiplayer Online Game (MMOG) with a range of distinct areas with different UD patterns. Through a broad analysis of the datasets we built from this, we draw out key insights and observations that delineate where the relevant constraints and requirements lie. We then use this to inform the design and implementation of our UD system, such that these requirements are met within the existing constraints.

With this chapter, we aim to answer a number of questions that reveal these requirements:

1. How is activity distributed across areas?
2. How dense do Area-of-Interests (AOIs) get?
3. How many players / how does player activity change over time?

4. How does player count relate to AOI density across different areas?
5. How are players distributed within areas?
6. How active/idle are players across different areas?
7. Are there different player mobility classes within the same area?
8. How prevalent is uniform motion across areas?
9. Do players exhibit group clustering?
10. What kinds of cheating are common?
11. Can vote-based ranks / social networks be used as a proxy for "reputation" to be used in reputation-based P2P neighbour selection? Can other metrics (e.g. account age) for evaluation purposes?
12. Is device resource heterogeneity significant?

Further to this, we must acknowledge the additional constraints posed by our deployment environment: the browser. These constraints will drive our design and evaluation beyond just the NVE constraints. To capture these, we seek to answer two additional questions:

13. What is the connection establishment time overhead with respect to pairwise latency?
14. What are the limits to the number of connections across browsers/devices?

We group these questions into different sections, explain why they are important, and answer them empirically. Wherever we tackle a different question, we will signpost this through repeating a question in a blue box. From our results, we draw out design and evaluation requirements which we put in green boxes. At the end of this chapter, we collate these and summarise.

3.2 Existing datasets

In the same way that mobility models can be developed from real mobility traces, some P2P Interest Management (IM) approaches directly use mobility traces for evaluation

[132, 38]. These traces are usually collected by strategically placing bots in Virtual Environments (VEs) and logging all position updates they receive. The virtual environment of choice in literature is overwhelmingly the social MMOGs Second Life, although some papers build their own mobility dataset from World of Warcraft or the First-Person Shooter (FPS) Quake and its sequels, or other lesser known games such as Freewar.

It is important to note however that position updates may be significantly sparser than what the player actually sees, and clients will usually at the very least interpolate positions. Ever since TRIBES implemented rollback and client-side prediction for player locations in 2000 [39], virtually every serious networked game has done the same. Position prediction (in addition to interpolation) can then lower inconsistencies between peers even further.

At any rate, these logs of player locations over time can then be played back over real or simulated networks and the metrics discussed previously can be measured. After reviewing as many instances as possible of traces like these that were collected in the literature, we have reached out to the authors involved. At the time of writing this thesis, there are no public location trace datasets available. Any datasets that were at one point publicly available (some even used again in other papers by different authors) are now defunct. Of the authors who have responded to us, in most cases, these no longer have access to the traces they used, for a variety of reasons.

At the same time, video game publishers have no incentive to build or release datasets like these, as the research that would make use of this data is very specific, exploratory, and difficult. Small publishers do not have the resources to undertake this research, while big publishers do, but might as well focus these resources in more predictable, well-established directions (such as simply paying for more/bigger servers).

For these reasons, we have decided to build our own dataset. This dataset can be used for a variety of different kinds of research, e.g. on mobility, P2P topologies, game AI, social network analysis, even online linguistics (from game chat data), among many other areas. In the following sections, after briefly describing this dataset and how we built it, we focus on mobility in the context of IM and P2P topologies that take avatar virtual positions into account.

3.3 Our dataset

Some general statistical mobility datasets are currently still available, however this kind of evaluation requires full avatar location traces. At the moment, this means that in order to

perform a trace-based evaluation, we must collect this data and build a dataset ourselves. In an ideal world, several complete, high-quality datasets would exist for different games and game types that researchers could benchmark their algorithms against and compare results in a robust way. We attempt to bring that closer to reality by compiling a rich dataset of network and mobility traces for the browser-based NVE *Manyland*. This is one of the major contributions of out work.

3.3.1 Overview of Manyland

The self-described "sandbox universe" *Manyland*[1], is a browser-based MMOG where players collectively build the worlds in which they play. Besides creating new areas and objects, players can also script behaviours within these, making it almost like a game with a built in game engine, so different areas can have wildly diverging genres of gameplay — from fast-paced shooter areas to social hangout areas. Figure 3.1 is a set of gameplay screenshots from across different Manyland areas.

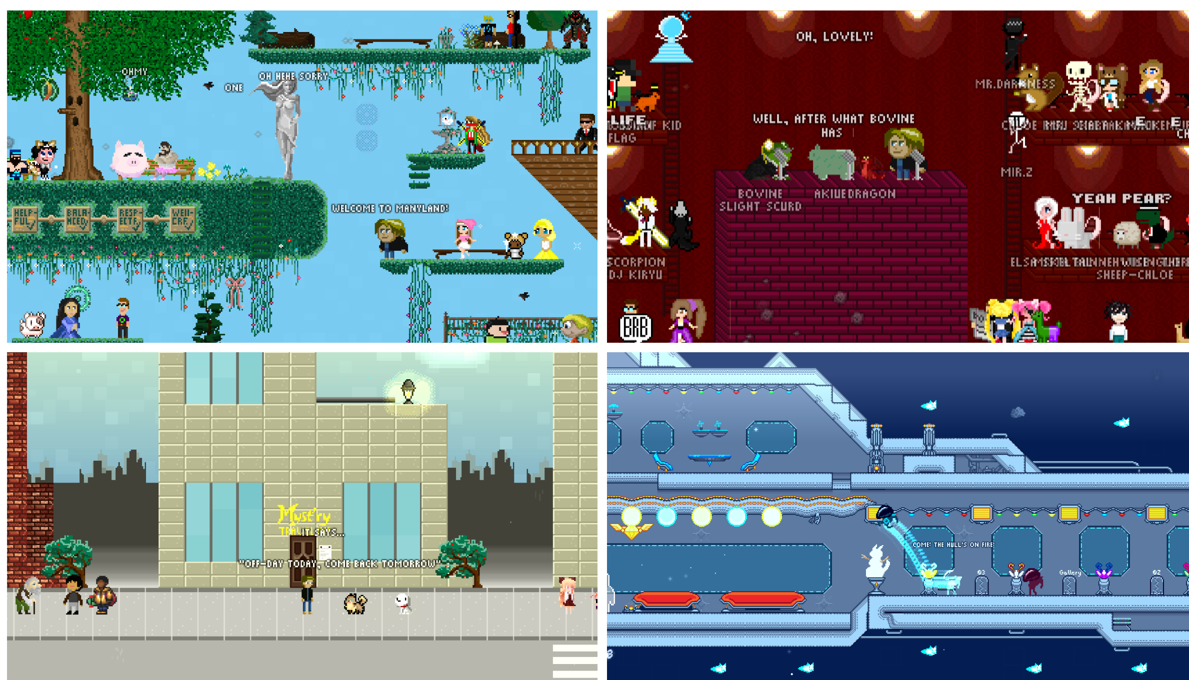


Figure 3.1: A set of gameplay screenshots from across different Manyland areas, courtesy of the official Manyland press kit[2]

Besides this, Manyland has a number of very pertinent characteristics that make it an ideal NVE to examine in the context of our research. Indeed, datasets from other NVEs

[1] http://findmanyland.com
[2] http://www.findmanyland.com/press/

would lack several of these. These characteristics are as follows.

- While clients exist for other platforms (like iOS or Steam), these are simply web-browser wrappers, and Manyland is browser-based at its core
- The universe is divided into user-created "areas" that can have wildly different kinds of mobility patterns and network requirements. Areas can extend infinitely in all directions. Some are sprawling, while others are small and contained. Some are more conducive to socialising, while others are for competitive games with a high degree of motion. Some even change the mechanics of the game from a platformer to a top-down view.
- Manyland has a built in friend network and reputation system driven by upvotes and flag reports (formerly downvotes). Users can be ranked between 1 and 5, which also gives reputable users more moderation privileges. Users with rank 10 are admins, and users with rank 0 are banned. This built-in reputation system is invaluable for investigating the use of reputation systems in neighbour selection and its effects on mitigating cheating behaviour in a network.
- Recently, the number of players online per day has sunk by an order of magnitude. The official website boasts 1000s of concurrent players daily, but as a result of growth being stifled by costs, this number fluctuates in the 100s today.

The last is unfortunately a struggle felt more strongly by many independent multiplayer game developers. Unlike single-player games, multiplayer games have running server and maintenance costs that can scale disproportionately with the number of users. Many independent developers choose not to relinquish integrity and quality in exchange for more aggressive advertising and monetisation strategies, and as a result fight an uphill battle.

3.3.2 Data collection

Like many other browser-based multiplayer games, Manyland uses WebSocket connections for communication between the server and clients. Clients send updates to the server, and the server then forwards these to players within the client's AOI (with a few exceptions). These messages are obfuscated and compressed (also called "minification"), however this is straightforward to reverse engineer, as the deobfuscation and decompression are part of the client.

With the permission of the Manyland developers, we built bots that passively record all the traffic with a wide coverage of the most popular areas. In order to keep our bots lightweight, we do not run a full client. Instead, we create our own stripped-down client in Nodejs to circumvent the large overhead of headless browsers. The process can be summarised to the following steps:

1. We first make a series of three different requests to the main page, simulating browsing to it, in order to get session cookies
2. We then query the latest minified client source code, safely `eval` it, and extract the functions we need based on their location relative to parts of the code that remain unchanged, and deobfuscate these
3. After parsing our previously collected session cookies, we set an `X-CSRF` header to authorise all subsequent requests
4. The next steps involve doing the bare minimum required to pretend to be a real client; we send a series of initialisation requests to undocumented API endpoints
5. The response from one of these gives us the WebSocket host and port that we need to connect to, as well as other parameters
6. We connect to this WebSocket and send a series of initialisation messages down it, interleaved with delays
7. These bots cannot be invisible, so in order to have them blend in as much as possible, we send a series of messages to the server to (i) change our name to a one picked from a list of realistic fake internet handles, (ii) change our appearance to a random, generic, community skin, and (iii) set the state of our player to “sleep” so they are effectively ignored for being Away From Keyboard (AFK)
8. To position our bot, we send a spoofed position update to the WebSocket server as defined by the script that orchestrates these bots. In a different context, doing this step would definitely be considered an exploit
9. The bot is now ready to passively collect any and all inbound application-layer traffic on the WebSocket connection (with only occasional heartbeat messages outbound to prevent despawning)
10. All data is anonymised on collection (details below)

These bots are spaced out in a grid formation in such a way that they have full AOI coverage of the entire area, with a level of overlap that makes every second bot redundant in case of disconnections. For areas that extend indefinitely, we have the bots on the edges fork another bot process next to themselves when they detect any player in their AOI, such that we capture everything. We store a list of bot user IDs that we use to later filter the bots out of the collected dataset, and we also remove duplicate traffic.

Furthermore, a lot of game logic (e.g. game physics) is entirely client-side, with no server-side validation. Since our lightweight bot client does not simulate physics at all, we make sure that the bots are placed in positions that would not raise suspicion (e.g. by defying gravity).

The actual updates, after unminifying them, are JSON objects. State updates are binary, while other updates are plain text but both are parsed and saved as JSON objects. Each has a property that says what type of update it is (e.g. a chat message, interaction, state update, etc). There are currently 64 types of updates, and the schema of the object will depend on the type (e.g. position updates may have `x` and `y` properties).

The raw traces are simply streamed into files (a new file per day) where each line contains a timestamp, the deobfuscated update type, and the entire JSON object (as a string). We create many other derived datasets from this, but for this chapter, we mainly care about mobility. To make this more usable for our purposes, we therefore also convert these full traces into stripped-down mobility traces.

The mobility trace format is simple. Each trace file is transformed into a CSV file where the columns are: timestamp, player ID, x coordinate, and y coordinate. The first time a player ID is seen is considered a spawn event. Any line without an x and a y coordinate is considered a despawn event.

The questions we laid out earlier also call for the analysis of other kinds of data, such as reputation data and other statistics. For these, we also built a set of scripts and crawlers to gather other metadata and statistics of the game. We explain these in more detail in the relevant sections.

We ran these bots sporadically through 2018 and 2019, as we were refining them, and then continuously ran the most recent iteration since mid-October 2019, and have been running them since. The server that runs these bots has a `cron` job for compressing collected traces into tarballs on a weekly basis, copying them to a different server with more disk space, and deleting them off of the server.

We make the data that we have gathered since mid-October 2019 available, in tarball form, post-anonymisation, on request for research purposes. With the exception of a

measurements that we ran across the entire dataset, the majority of our research concerns itself with only a subset of this data, which is much more manageable is size and freely available.[3] Player IDs and other identifying information has been anonymised and chat text has been censored to protect the privacy of the players.

3.4 Areas and workloads

One of the biggest challenges in designing P2P systems for NVEs is ensuring that it can support different genres and environments. The technical requirements for a large-scale social MMOG are vastly different to small-scale, fast-paced FPSs. Indeed, while some previous work does consider a range of workloads like these for evaluation [9], most get around this problem by simply limiting their systems to one kind of environment, such as Second Life-like social MMOGs. In §5, we evaluate how our system fares under different workloads like these, however we must also consider different genres at this stage, as they will directly influence the requirements we derive from them.

We note that an analysis of data collected from Second Life by Varvello et al. [133] showed that the distribution of players across areas is very skewed. 30% of Second Life Regions are consistently empty, 45% consistently have 5 or less avatars, and only 5% of Regions reach 30 concurrent avatars as a maximum. It is not uncommon that players concentrate around hotspots or popular areas in social MMOGs, in which case traces from just these popular areas are more than sufficient for evaluating P2P models.

Previously, we have also highlighted that one of the advantages of using Manyland as a source of mobility data is the large variety of areas; from social hotspots, to fast-paced player versus player (PvP) combat areas. To clarify, these areas are not discrete parts of a larger continuous space, but rather akin to different rooms/shards/dimensions hosted by different servers. The environments across areas are entirely different and players cannot directly interact across areas. It is also not possible to get from one are to another simply by walking (in fact, areas can extend infinitely in all directions) but only by using a portal-like door, teleporting there with a command, or browsing to a different URL path. We must therefore select the areas for analysis carefully to capture both high activity and workload variety. To do this, we ask:

How is activity distributed across areas?

[3] `https://www.kaggle.com/dramar/manyland`

We have found that Manyland areas share similar behaviour, and our analysis confirms what is already held true by literature. Manyland currently has over 720,000 areas with 100-200 new areas being created daily. We have crawled 200,000 of these and the resulting statistics have proven this sample representative. Beyond the first few thousand areas, the distribution remains largely unchanged.

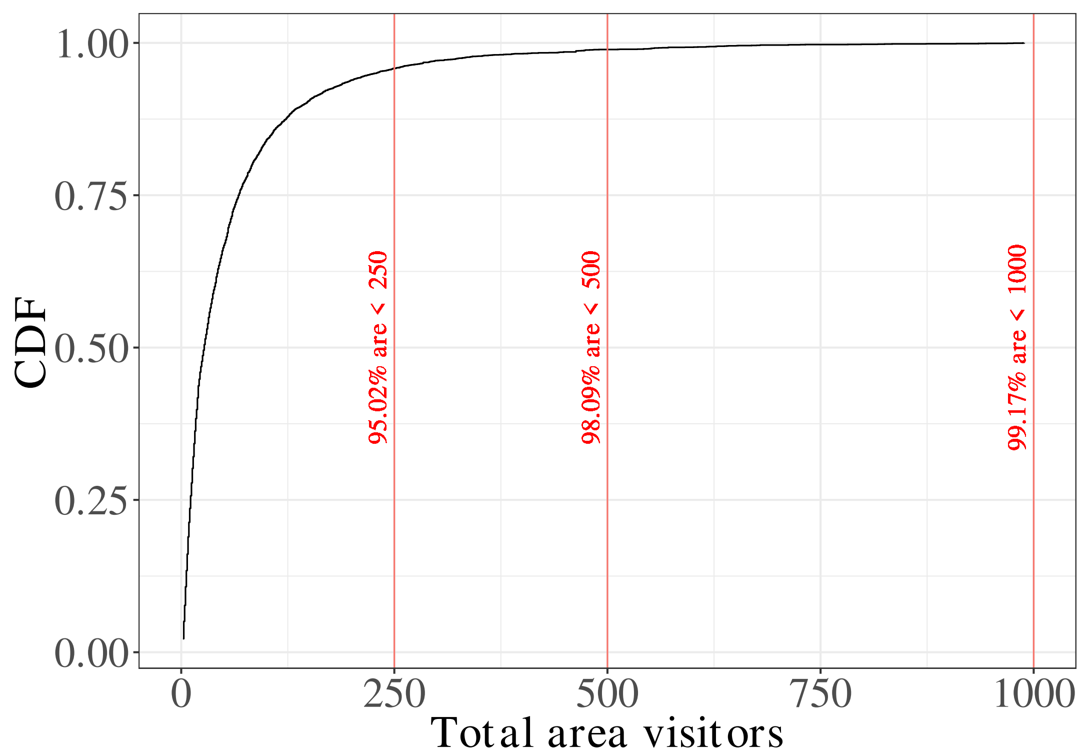


Figure 3.2: An empirical cumulative distribution function of total area visitors ([0, 1000] for visualisation)

Figure 3.2 shows a CDF of total number of visitors per area. We limit this plot to 1000 total visitors on the x axis for visualisation purposes and because over 99% of areas have less than 1000 total visitors.

The area we sampled with the largest number of total visitors in its history had 749291 visitors. This area, and other outliers like it, pull the mean to 479.7 visitors. However, the median is 29 visitors, and the third quartile is at 71 visitors. The first quartile is similarly low at 12 visitors. The minimum is 2 visitors; the first is the creator of the area who automatically visits the area when first creating it, and the second is our area crawler.

Previous work analyses mobility within a couple of the most popular areas in Second Life. Here too we select the two most popular areas for subsequent analysis. Not only is a higher number of users and activity more useful for setting the requirements for scalable

systems, but these are essentially representative of other areas in the game. Areas with very little activity are trivial to handle as a small number of players can simply be completely connected, or indeed areas with a single player need no P2P networking at all.

We go a step further by selecting two different popular areas with vastly different characteristics, similarly to related literature [9]. Like in Second Life and similar, it is typical for activity to be focused in a small number of areas as we have seen. Fortunately, the two of these that we focus on are such areas and represent two different "genres" as luck would have it.

The first, which we will refer to as the *dynamic* area, is what is known in-game as one of the public "outer ring" areas. New players are most likely to spawn here, and the area sprawls so far, that it literally takes days of non-stop walking to reach the furthest stretches of where players have built. Players tend to cluster around the centre of this area where new players spawn and due to the transience of what is built in this area, players tend to wander and explore.

The second, which we will refer to as the *static* area, is a closed building area with a number of rooms. The layout is unchanging as only the original creator can edit it and the original creator has not been online in years. Over time, it has become the most popular area for chatting and social interaction, primarily due to its lack of moderation. Generally, players sit in one of the rooms and chat with each other, occasionally moving to different parts of the building.

We focus on these two popular areas in particular as they illustrate two extremes — slow-paced, social workloads as well as fast-paced, motion-heavy workloads. A player in the static area has an average position update emission interval of approximately 3.4 seconds, while one in the dynamic area has an average interval of 700 milliseconds. This rate is not fixed and position updates are sent on motion. We will explore this further shortly.

These are also the most popular areas of their kind and provide the most useful insights due to their relative levels of traffic. We highlight two future design requirements:

Design requirement ①
Topology must perform well under static conditions

Design requirement ②
Topology must perform well under dynamic conditions

3.5 Measuring AOI density

In §2 we discussed IM and AOIs. Intuitively, the number of players within an AOIs will drastically affect the performance of any given topology. For example, a topology that completely connects AOIs will scale badly for dense AOI. Meanwhile, one that leads to heavy intra-AOI forwarding may be more prone to loss and malicious behaviour. It is therefore important that we understand how dense AOIs get, and how this changes over time (if at all), such that we can find topologies that accommodate this.

3.5.1 AOI density due to player activity

How dense do AOIs get?

To answer this question, at a granularity conducive to useful insights, we play back a week-long segment of our static and dynamic traces, from 2019-11-11 to 2019-11-17 inclusive. At ten minute intervals, we go through all players and find the mean density of their AOIs (i.e. how many other players there are in their AOI). Figure 3.3 plots these over time for the two traces. The trend lines, with 0.95 confidence intervals, are computed through locally estimated scatterplot smoothing (LOESS).

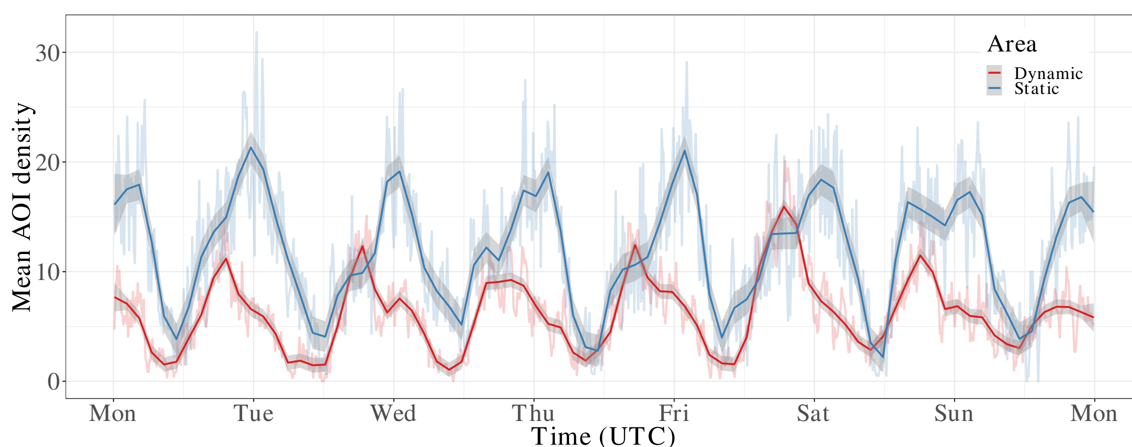


Figure 3.3: Mean AOI density across areas sampled at ten-minute intervals

The most striking aspect of these measurements is the seasonality — we see very obvious diurnal cycles that persist across the weeks, months, and years, with only a secondary, slower movement based on Manyland's popularity over the years. It is not a stretch to presume that the changes in AOI density could strongly correlate with the number of players online at any given time. We therefore ask:

How many players / how does player activity change over time?

As we have previously alluded to, Manyland has seen a stark decline in the number of players since when it used to be in the thousands. Still, the level of activity we see is similar to other social MMOGs like Second Life which most related literature analyses. In order to understand how player activity affects other metrics over time, we make use of undocumented Manyland API endpoints to sample global player activity at ten-minute intervals. This is more accurate than our bots, especially since our bots do not have complete area coverage, and indeed serves the secondary purpose of telling us what percentage coverage our bots do have.

In addition to this, we also track the players online in our two sample areas (static and dynamic) which, out of over 720,000 areas, account for 33.69% of online users combined. Figure 3.4 is a plot of these samples over time.

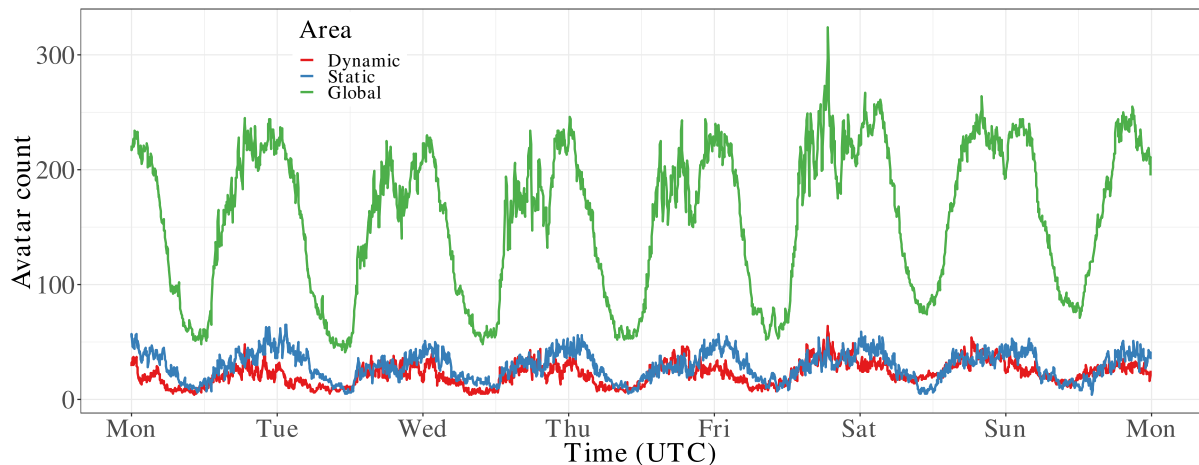


Figure 3.4: Mean player count across areas sampled at ten-minute intervals. The global avatar count (green) is included to illustrate the proportion of players in the two areas of focus compared to game-wide

The seasonality of this data is immediately obvious and we can see how it correlates with AOI density. We have clear cycles of activity that correspond to the hours where most players are active. There are increases on Fridays and weekends, which makes sense, as that is when most players are off school/work. This seasonality is important, since topologies need to not only be able to scale to different numbers of players, but also handle the transitions between these numbers well if they are not constant. If we focus on the pattern for just a 24hr period, we also see a subtle bimodality. We summarise the distribution of this data in table 3.1 and plot these in figure 3.5.

Table 3.1: Statistical summary of player count distributions across environments

Area	**Min**	**1st quartile**	**Median**	**Mean**	**3rd quartile**	**Max**
Dynamic	4	15	22	22.58	29	64
Static	4	20	30	29.7	39	65
Global	41	97	173.5	159.9	216	324

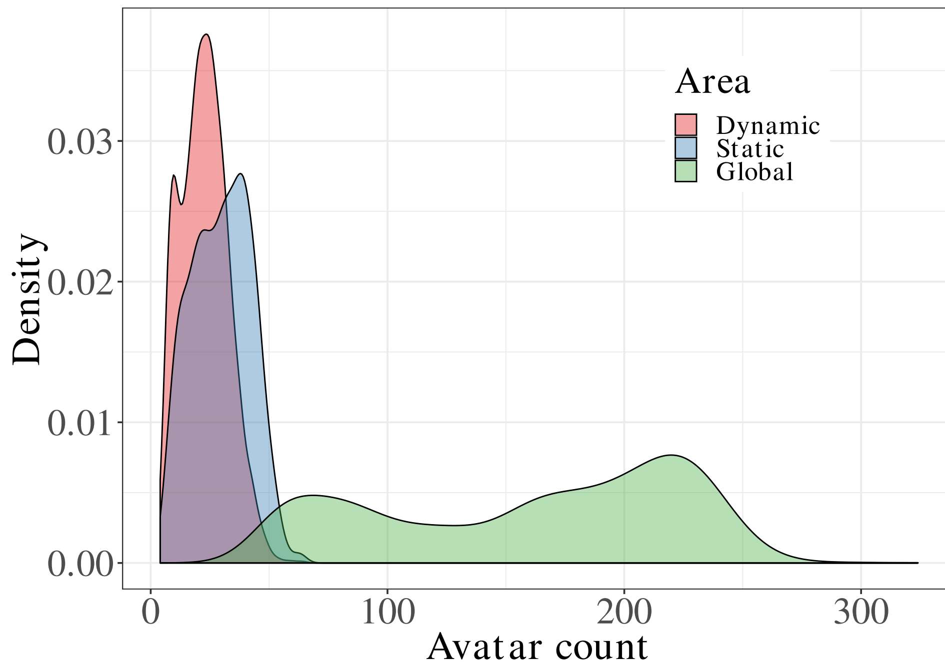


Figure 3.5: Density plots of player count across areas

Here, the bimodality becomes much clearer. One of the developers of Manyland gave us some insight on the demographics of the game (more up to date information in §3.9). The overwhelming majority of players are based in the US, followed by Canada, and the UK. Some countries (such as Zimbabwe) have only one player.

All times are in UTC, so if we take US timezones as being between UTC-8 (San Francisco) and UTC-5 (New York), the second, larger bump roughly corresponds to their late afternoon and evening (17:00 – 20:00). This bump is itself mildly bimodal, which can be explained by the concentration of people towards the east and the west of the US in contrast to the middle. Similarly, if the EU (UTC, UTC+1) corresponds to the first bump centered around 18:30, which fits perfectly.

Interestingly enough, there are peaks in traffic at noon on school days. According to the developer, this can be attributed to the rise of Chromebooks (which are also generally weak "disposable" laptops) as the standard school computers. Manyland was featured

on the Google Chrome Web Store and also won a “best of 2014” award, which built a lot of momentum for Chromebook activity and spread in schools by word of mouth. When the noon lunch break starts, many school children visit Manyland. This also sank the average player age.

It is conceivable that one can infer a coarse geographical location of any given player reasonably accurately after tracking their online activity for a sufficient length of time, knowing the average activity patterns by timezone/latitude and distribution of players by country. We aim to augment our dataset with anonymised ground-truth user geolocation data such that this kind of research can be performed with it.

Diurnal patterns like this are very common in online games. It is important to consider spikes caused by virality and account for dynamic load as the number of players change relatively predicable over the day, week, and beyond (e.g. the summer).

Having seen shown the correlation between player count and AOI density however, we are left with a much more crucial question.

How does player count relate to AOI density across different areas?

To answer this, in figure 3.6, we plot AOI density against player count across our two environments. Here too, these are trend lines, with 0.95 confidence intervals, that are computed through LOESS, but with a span of 0.5 (as opposed to 0.05) for maximum clarity.

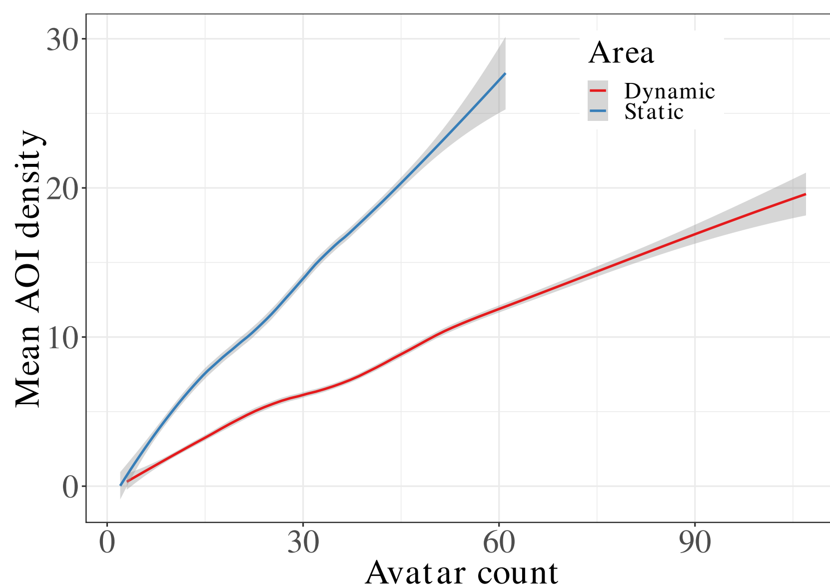


Figure 3.6: AOI density versus player count across environments

Here we make a critical observation: while AOI density will increase linearly with the number of players, the rate at which AOI density increases with player count strongly depends on the type of environment. Recall that the "static" area is smaller, more contained, and players generally sit and chat in close proximity, while the "dynamic" area is larger, sprawling, and has a significantly higher degree of motion. It makes sense therefore that in the static area, a marginal increase in mean AOI density by 1 requires only approximately 2.25 new players, while in the dynamic area, it would require approximately 5 new players. The static correlation has over double the gradient!

There is an important caveat here however. The static area has limited space, and it is unrealistic to assume that player count can increase indefinitely causing AOI density to increase in turn. If an area is too crowded, it becomes impossible to make any meaningful interactions with others. Players naturally give each other space when chatting, especially because otherwise chat text (which appears over a player's avatar) would overlap and become unreadable. We posit that there is an upper limit to AOI density that is dependent on area size, but this is difficult to observe in-game and prove because there would be no consistent plateau, rather the area would just become less popular and people would disperse to other areas.

The results in this section lead us to draw out several additional design and evaluation requirements:

Design requirement ③
Any Application-Layer Multicast (ALM) must accommodate a range of AOI densities

Design requirement ④
Topologies cannot assume constant activity/density and must accommodate cyclic changes over time

Evaluation requirement ①
Area size and number of players in a synthetic trace must be set in such a way that realistic AOI densities are emulated

Design requirement ⑤
Topology must scale to a minimum typical volume of players (replication, connection limits, etc)

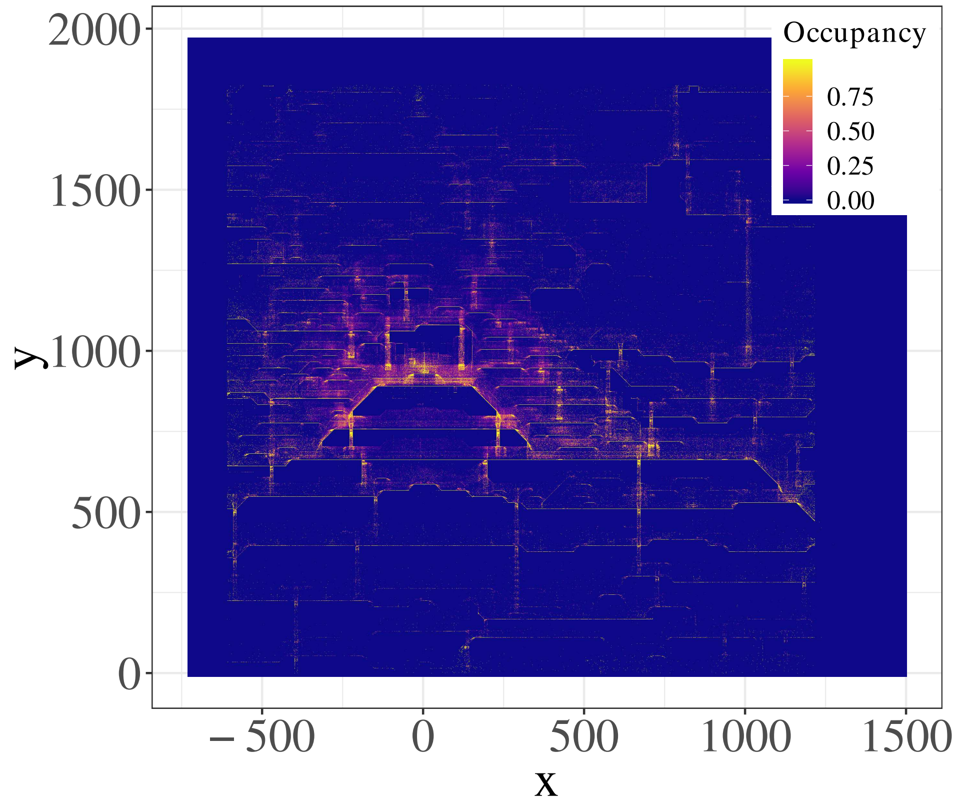
3.5.2 Hotspots

In comparing the characteristics of different types of areas, we have shown that existing work on general-purpose NVE networks that uses only social NVEs for evaluation is at risk of being overly circumstantial. It does not end there however. State-of-the-art synthetic mobility simulations always recognise that players do not distribute themselves uniformly across a scene. Obstacles (such as walls), physical features of the scene, and game-related hotspots all create uneven distributions of occupancy. This will of course affect topology churn since these hotspots are like dense mini-areas nested within sparser areas. So far, we have only looked at the *mean* AOI density over time, which ignores this notion. We therefore measure *occupancy* to answer the question:

How are players distributed within areas?

To measure occupancy, we divide the area into cells. We define occupancy as the per-cell sum of time players spend in any given cell across our week's worth of mobility traces. This sum is across all the players in a cell (i.e. a cell with 2 players for 2 seconds has an occupancy of 4 seconds).

It is important to note that the way we measure this does not consider intermediate cells, as motion is discrete. In other words, if our traces log an avatar's position as $(0, 0)$ at $t = 0$ and then $(0, 2)$ at $t = 4$, then the cell at $(0, 0)$ has had an occupancy of four seconds, but $(0, 1)$ (which the user had to physically cross to get to $(0, 2)$) remains unchanged, assuming a cell size of 1×1. This will not be ideal for all use cases, but for the purpose of highlighting the existence of hotspots and waypoints, it is sufficient.



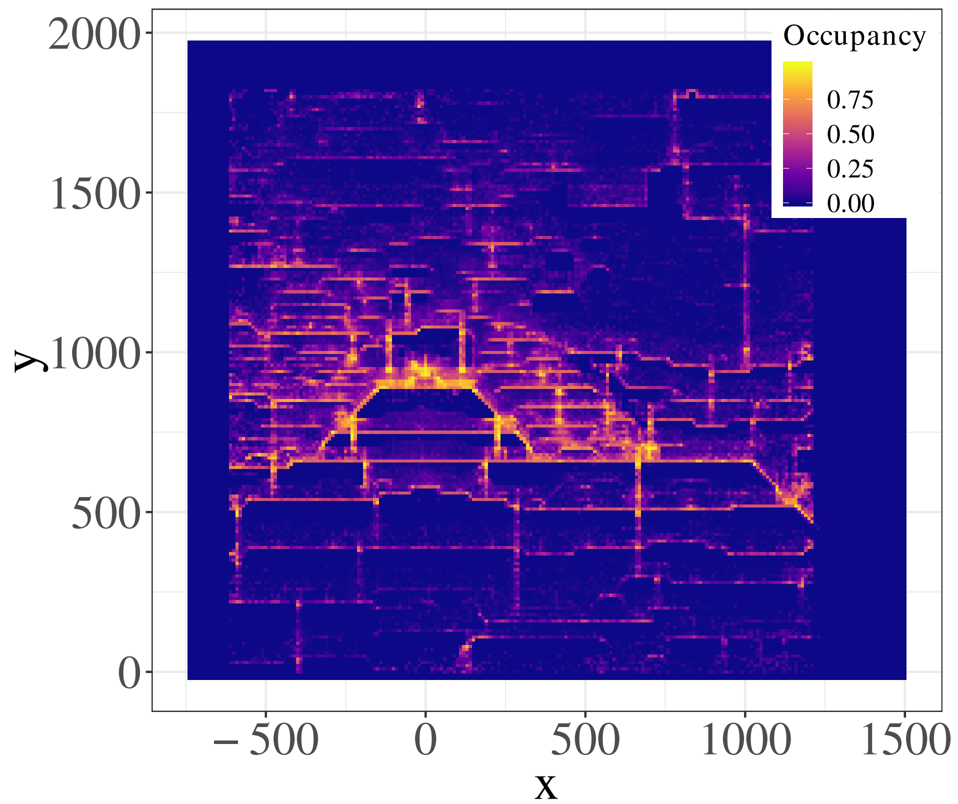
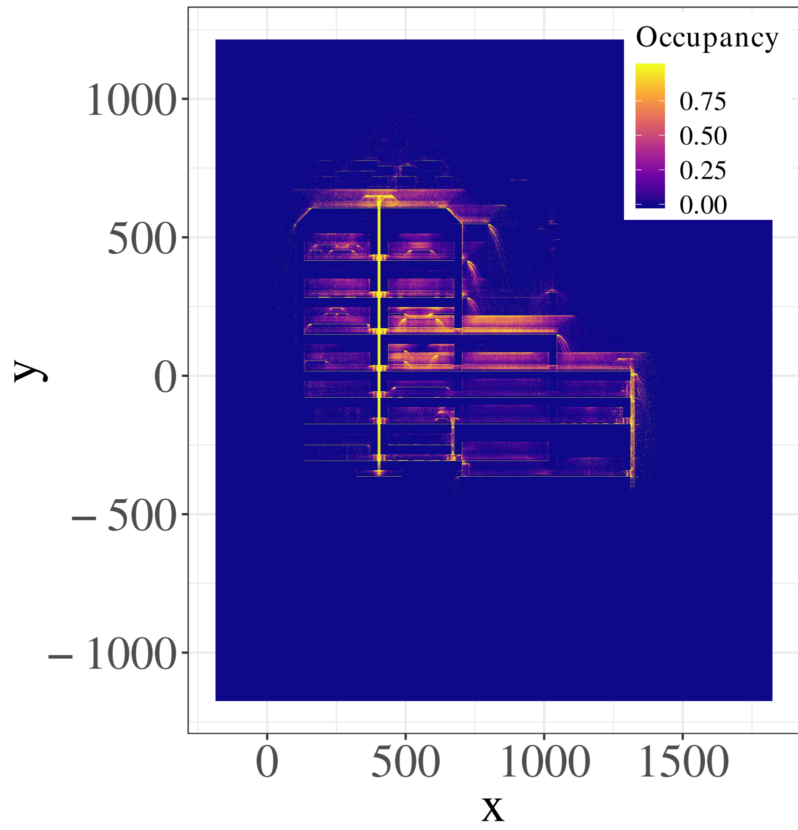


Figure 3.7: Occupancy heatmaps of the *dynamic* area with cell sizes of 1×1 (top) and 10×10 (bottom)



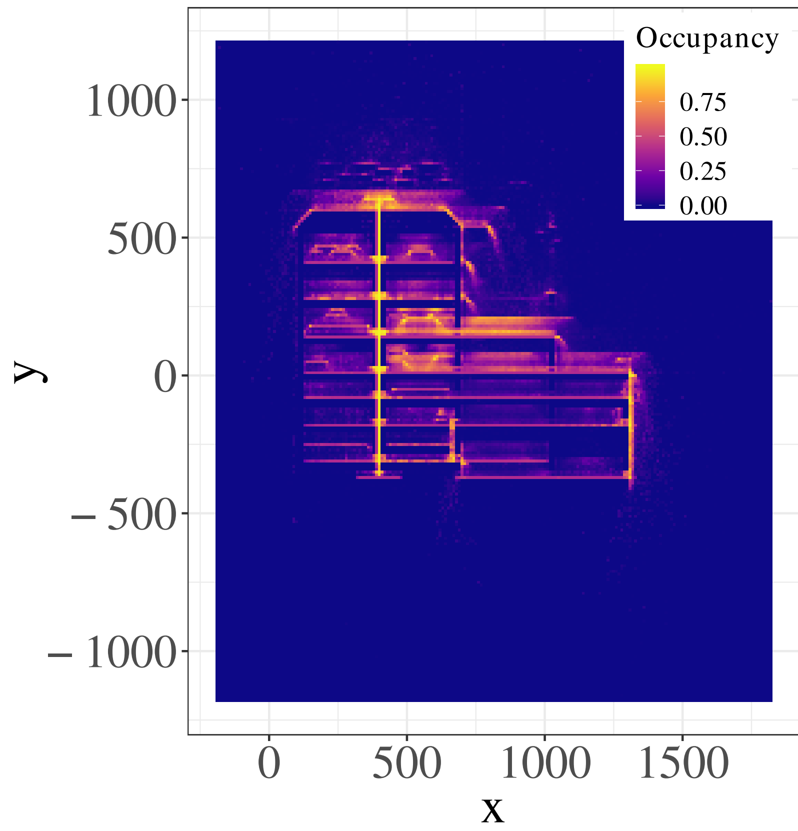


Figure 3.8: Occupancy heatmaps of the *static* area with cell sizes of 1×1 (top) and 10×10 (bottom)

Figures 3.7 and 3.8 show heatmaps for both dynamic and static areas respectively. For each, we show heatmaps with cell sizes of 1×1 (top) and 10×10 (bottom). In order for these heatmaps to be useful for visualisation, we include only the 99.5^{th} percentile. If we do not cut out this top 0.5% of extreme outlier, the heatmap is flat with a few spikes. These spikes are not useful to us, as they do not represent normal player behaviour. These are bots, idle game windows that are minimised/forgotten, or players where the server did not notify clients of their disconnection. Our own bots are filtered out of the traces.

We note that these heatmaps are global and do not tell us anything about per-player behaviour, for example if one person moves a lot and others move very little. This is something we expand on in the coming sections, however there are still insights to be gleaned from these globally compiled measurements.

We also scale occupancy logarithmically for similar reasons. Where o is our raw occupancy, we transform this through $1-e^{-ko}$. k is a constant that controls the steepness of the curve. We tune this to scale up the brightness of the dimmer areas, or seen differently, to "flatten" the high occupancies that dominate the colour scale, for the scale to be more useful. Table 3.2 summarises these parameters.

Table 3.2: Additional occupancy heatmap information

Area	**Cell size**	**k**	**Heat mean**
Dynamic	1×1	5	0.0267100
	10×10	5	0.0689100
Static	1×1	5	0.0288300
	10×10	5	0.0407494

In the case of the dynamic area, we have a very clear pattern of higher occupancy at areas nearer the spawn point (at around $(0, 1000)$) with smaller hotspots throughout. Paths that are well traveled show the highest occupancy, while paths that have many alternatives less so. This is in contrast to the closed static area where occupancy is highest at the most popular rooms and the paths between rooms. Occasionally avatars may glitch into the wall, intentionally or unintentionally, causing a gap that is otherwise empty (because a wall occupies it) to have some avatar occupancy.

These figures serve to show primarily that occupancy is far from uniform. They also show in what way they are non-uniform, with a high degree of clustering around hotspots, and very little occupancy between these. From the analysis carried out in this section, we draw out an additional evaluation requirement based on the information we now possess:

Design requirement ⑥
Topology must account for concurrent sparse and dense regions

Evaluation requirement ②
Synthetic workloads must emulate realistic occupancy distributions

3.6 Measuring topology churn

So far we have considered the topological implications based on spatial characteristics of NVE traces. In this section, we consider temporal characteristics in more detail.

After briefly examining session lengths, we focus specifically on motion. It makes intuitive sense that the extent of avatar motion will affect AOI churn and therefore — assuming that the P2P network in question seeks to maintain optimised topologies — topology churn. For example, a static environment can stick with the same topologies for longer, as AOIs do not change as frequently. This is especially important in a browser environment as the connection/disconnection of links between peers carries a significant overhead.

So while we have previously shown that the dynamic environment has the benefit of lower AOI density overall, these benefits may be counteracted by high topology churn due to increased motion. Because of this, we first seek to characterise player motion across environments at a high level.

3.6.1 Session length

Topology churn is not just caused by motion, but also the rate at which players enter and leave an area (or log in and out entirely). It is important to consider this both when building a P2P system but also when generating synthetic traces for evaluation. We measure the length of sessions across players in our dataset areas in order to get a better idea of how this plays out in reality.

How long are players online at a time?

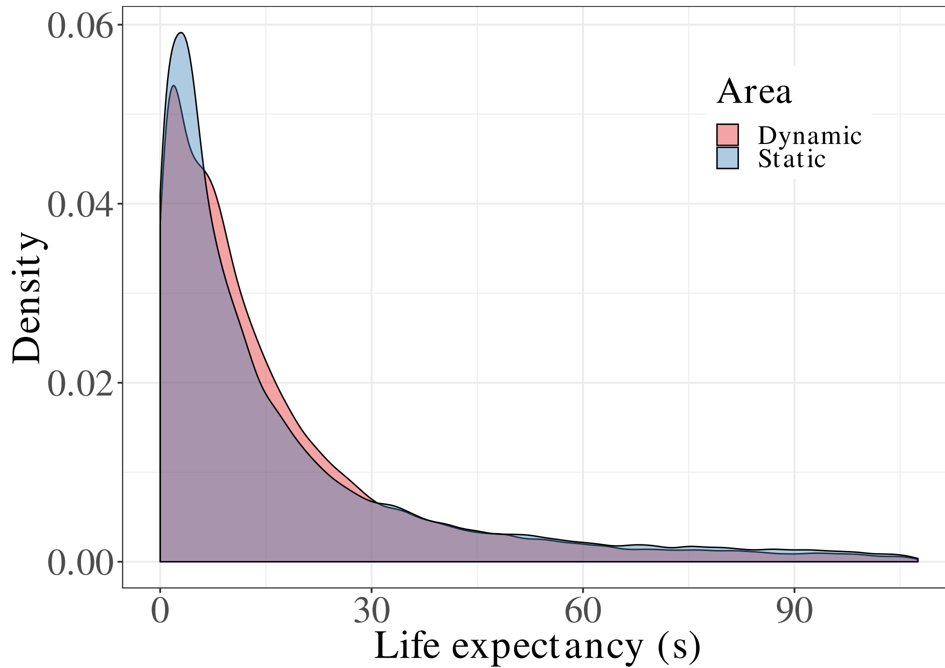


Figure 3.9: Density plots of player session lengths across areas, capped at the 90th percentile for readability (overlap of both is purple)

Figure 3.9 shows distributions of player session lengths across areas. Here, the most striking observation we make is how prevalent very short-lived sessions are. Not only are these prevalent, but this prevalence is mostly agnostic of the type of area.

In the dynamic area, we have a median of 13.16 second sessions, and in the static area 10.233 seconds. This is likely to due to the fact that both of these areas act almost as central "hubs" due to their popularity. Players log in, only to immediately travel to a different area. Similarly, players may briefly visit these areas to see who is there and decide not to stick around. This behaviour is prevalent across any popular area, likely across any game, irrespective of the type of area. The places with the most peers will also have the most short-lived sessions.

At the same time, we do see a very large range of session times. The longest session at the dynamic area was just over an hour and eight minutes long, and in the static area just over an hour and two minutes. Meanwhile, the mean session time at the dynamic area was 7.71 minutes, to the static area's 58.86 seconds. While these density plots might make it look like the sessions are very short, this is simply because there is less spread at short times and a high spread at higher times. A session that is 2 seconds long is double a 1 second session, but considered similar, while a 30 minute session is double a 15 minute session, but quite different. There are many players who do persist beyond a minute in both areas.

The main takeaway here is that these sessions can be very ephemeral and this churn can affect the stability of the network. For example, peers through which many paths with no alternatives pass (no redundant paths around the peer) will disconnect the network when they go offline. Based on these measurements, and the lack of correlation between the prevalence of short-lived sessions and area type, we cannot rely on area type as hint for predicting session length, and therefore peer ephemerality. As expected however, we must account for the longevity of peer sessions when computing topologies, as rewiring network paths to go through a newly-connected peer, only for them to disconnect seconds later, has the potential to decimate performance. We draw from this two requirements:

Design requirement ⑦
Topology must consider high peer ephemerality

Evaluation requirement ③
Synthetic traces must reflect peer ephemerality

3.6.2 Idle behaviour

How active/idle are players across different areas?

Players are neither constantly still, nor are they constantly in motion. The ratio of these depends entirely on the VE in question. For example, a social VE (or a social area in a VE) may be more conducive to remaining still in one area and chatting. Meanwhile, an action game, such as an FPS, may have players moving near-constantly at risk of death. SOTA mobility models take this into consideration [91], and we must therefore characterise this in our dataset.

First, we classify avatars as *active* or *idle* over time. *Idle* players are only idle here with regards to motion — they can very well be chatting in-place. As part of our dataset, we do however make true "AFK" data available, as we track when players lose focus on the game window, for example by switching to a different browser tab. This manifests itself in-game as avatars closing their eyes.

As we are dealing with position updates, we need to find a time threshold beyond which players are considered idle if they do not update their position. This threshold cannot be too small, as this would result in noisy data with players switching state at a high rate, for example if they were simply slightly adjusting their position or turning to face another player (as opposed to traveling across the area). What we need is essentially

a high-pass filter to remove this high-frequency noise.

By finding occurrences of short burst of motion in our dataset, and manually classifying them, we have determined that a suitable threshold for this game that removes noise and aberrations is 3 seconds. If a player stays still for longer than 3 seconds, they can be considered idle. We do not add these three seconds back to a player's idle time, as doing so would merely shift the distributions 3 seconds to the right.

Earlier, we noted how a server-side bug can sometimes result in despawn updates not being sent even, after a player has disconnected, resulting in "zombie" avatars at the edges of the map. To avoid having these zombie avatars pollute our dataset, we only add to idle time measurements every time an update is received. Since these players never receive a despawn update after their final position update, the time they spent idle as zombie players is ignored from our dataset. Meanwhile, players that legitimately despawn are unaffected by this.

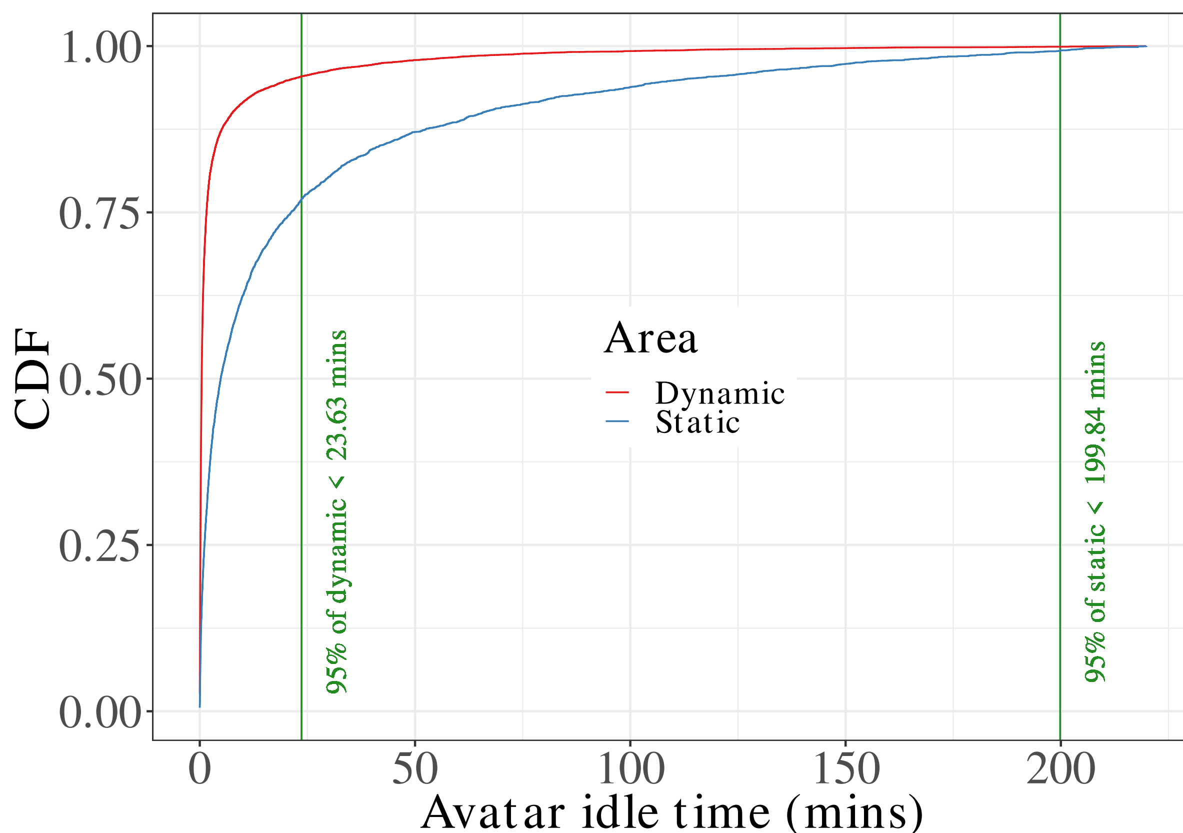


Figure 3.10: Empirical cumulative distribution functions limited between $[0, 220]$m comparing areas

Figure 3.10 shows empirical cumulative distribution functions for the idle time over players in the two areas we examine. We limit this plot up to 220 minutes as idle

times beyond that are outliers and simply make the plot less useful for interpretation. Indeed, we note that 95% of static area idle times are below 199.6781 minutes, while 95% of dynamic area idle times are below 23.63131 minutes. These distributions fit the characteristics of these areas, considering that the static area is much more conducive to chatting and socialising, while the dynamic area has much fewer idlers, and the ones that do idle have less reason to do so for as long.

More surprisingly however, 50% of static area idle times are below 5.58765 minutes, while 50% of dynamic area idle times are below 26.497 **seconds**. This is further evidence of the above observation.

There is clearly a large difference in how the type of environment affects topology churn. Related work that considers only the social MMOG case gets lower churn "for free" as a result of high idle times. When designing our P2P system, we do not have that luxury if we want it to be applicable to highly dynamic environments.

At the same time, the dynamic area has seen significantly more distinct/unique players over the week; 13,153 to the static area's 4,275 — an order of magnitude higher. Even with this high level of traffic, the sum of all idle times across all players is 59.53 days in the dynamic area, while it is 122.20 days (over double) in the static area. This is important, because it tells us that the type of environment implies more about the level of topology churn than the actual traffic it gets.

Table 3.3 summarises idle time distributions for both areas, converted and rounded from milliseconds to human-comprehensible units. This is across all users (not just those with one or more idle period).

Table 3.3: Statistical summary of idle time distributions across environments

Area	**Min**	**1st quart**	**Median**	**Mean**	**3rd quart**	**Max**
Dynamic	0ms	8.93s	26.50s	6mins 31s	1min 35s	1d 9h 11mins 4s
Static	0ms	1min 11s	5mins 35s	41mins 9s	26mins 5s	2d 1h 53mins 55s

The two minima of 0 milliseconds are likely as a result of players who came to the area just to check who else is there. They probably walked across the map and did not pause for longer than 3 seconds, before either going offline or to a different area. These statistics reflect what we know about these areas. We keep track of player idle times across sessions (for each player ID) so the maxima here do not mean that there are some obsessive players that were online on the order of days. Instead, this is the total idle time for that player across our sample week.

The most extreme player in the static area idles for over 7 hours a day on average. From what we have observed, these are players that usually leave the game open in a browser tab while they do other things. All this tells us is that this player had his computer on and browser running for at least 7 hours a day. Users like these are of course ideal for P2P networks!

This leads us to an important question. Can we generalise across players like this (like all related literature has in the past), or are there different "classes" of players? For example, excessively active versus sedentary and social players, in the same area. This is especially important to capture the requirements for generating synthetic mobility traces for evaluative purposes. These traces can be used to simulate player motion realistically while allowing us to explore extreme scale and patterns of motion to stress-test our P2P networks.

Are there different player mobility classes within the same area?

To answer this, we have every player that appeared in our sample week (across all sessions; short-lived and long-lived) keep track of their transitions between states. When we sample our measurements (every ten minutes like before), each avatar can either transition from *active* to *idle*, or vice versa, or remain in the state that they are already in.

The authors of Blue Banana [91] have done something similar in the past to design a more realistic mobility model. They built a state machine with three states: Halted, Exploring, and Travelling. They tweak the probabilities such that the number of players in each state at any given point in time roughly matches those in the Second Life trace. The difference between Exploring and Travelling is in maximum speed and the nature of the motion. Different kinds of motion are less relevant to us since we are less interested in generating natural-looking motion, and more interested in topology churn which can caused by any kind of motion whatsoever.

There are two problems with Blue Banana's approach in its applicability to us. The first is that they built this state machine only from Second Life traces, which is a social MMOG and liable to have similar characteristics to our static environment. The second is that it generalises across player classes. If there really is only a single player class, then this is fine, but otherwise too large an abstraction. With the following results, we aim to answer that question.

Our approach involves no guesswork; we calculate our state probabilities from our area traces directly. We sample state transitions at a granularity of 10 times a second

(100 milliseconds intervals) across the week-long traces. We sample at fixed intervals, rather than whenever an update is emitted, as we have measured that a player in the static area has an average position update emission interval of approximately 3.4 seconds, while one in the dynamic area has an average interval of 700 milliseconds. This rate is not fixed and position updates are sent on motion. Sampling at variable intervals would therefore make comparing the two environments invalid.

We sample at 100 millisecond intervals in order to capture high-frequency transitions in the dynamic area especially, otherwise the "resolution" of our data would make for incomplete plots rife with clear discretization error. However, this has the downside of making it seem like players remain in their current state with a much higher likelihood, when in actual fact, the higher the sample rate, the higher these probabilities will be. By definition, the more samples we have, the more of them will be between transitions.

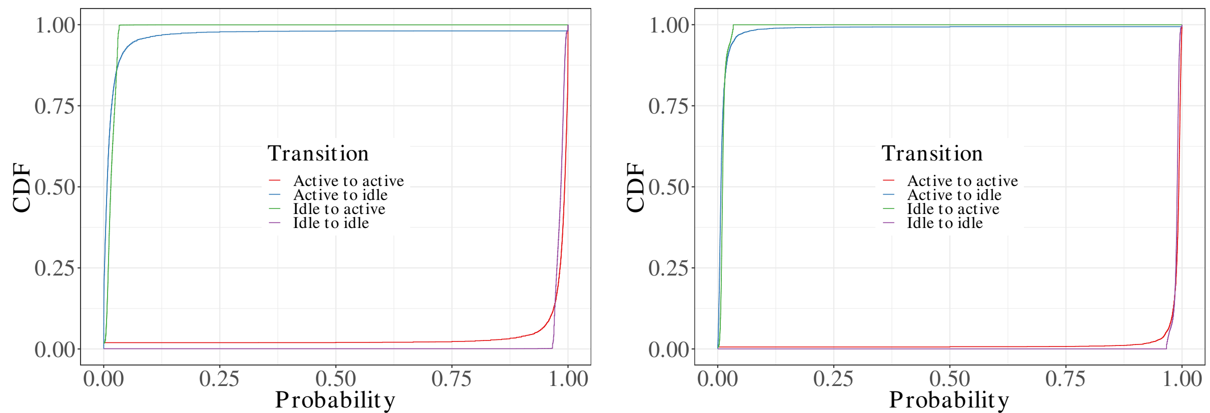


Figure 3.11: CDFs of transition probabilities across players for the dynamic (left) and static (right) areas

Figure 3.11 are CDFs of transition probabilities across players for they dynamic (left) and static (right) areas. Here we see a very clear pattern across both areas. The overwhelming majority of players switch states with at a very low probability, and tend to stick with the state they are already in otherwise. In other words, these states are very "magnetic" for all players. To clarify, at our sampling rate, if a player stays idle for one second, they have missed 10 opportunities to switch to an active state.

We can also see that the active-to-idle transition has a spike at the probability of 1.0, which the transition the other way around has barely any players with a probability higher than 0.04. This means that there is a small set of players that go idle almost immediately and stay that way. These probably do not stay online for very long or immediately teleport to a different area. If they do stay in the area, they likely come online, check who else is online, or check the in-game boards, and then leave. These are

then of course the same set of players responsible for the spike at probability 0.0 for the active-to-active transition (as active-to-anything adds up to a probability of 1.0).

These could in theory form an additional player "class", but it is fair to treat these are exceptions since all other players have very similar behaviour with a tight variance. We do note however that transition out of the active state have a higher variance in probabilities across players. This means that it is less predictable how long a player will stay active, and more predictable how long a player will stay idle. The reason for this could be that motion often depends on how far a player needs to travel, which can vary widely based on what their starting point and destination is. Meanwhile, idle times could be more more predictable (as we have also seen in figure 3.10) as they are location-independent, and indeed may be linked across players due to them being idle as a result of a joint interaction (e.g. chatting) that they end simultaneously.

Based on these measurements, we can (like [91]) create our own state machines here without losing important information. Figure 3.12 depicts these for the dynamic (left) and static (right) areas.

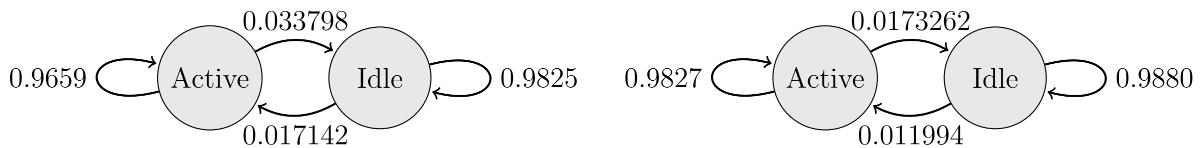


Figure 3.12: Mean player finite state machines for dynamic (left) and static (right) areas

All avatars start in the active state. As we have already seen, the transition between states is less likely in the static environment than the dynamic one. In the case of the static environment, once a player goes idle, they're more likely to stay idle, unlike in the dynamic environment.

At first glance, one might assume that these transition probabilities are quite similar and should result in similar player behaviour, unlike what we have observed with idle times being significantly higher in the static environment. We would like to stress however that it is the differences in probabilities that matter here and will ultimately decide which state is more prevalent.

It is absolutely imperative that results like these — if they are used for generating synthetic mobility data — take account of the sample rate. This is also why we did not round these probabilities to fewer decimal places above. Small differences here can have tremendous consequences. It is difficult to get right because, generally speaking, players do not emit positions at fixed intervals; they do so whenever the player moves and then at a rate based on how far they have moved since the last emission.

From this, we conclude that it is safe to treat all avatars as a single class in the context of mobility. However, different idle time patterns across different types of environments can have significant consequences on topology churn. The dynamic environment should be the worst case to consider. We are left with a set of design and evaluation requirements.

Evaluation requirement ④
Synthetic traces must emulate realistic active/idle states

Design requirement ⑧
Topologies must account for churn caused by dynamic active times

Design requirement ⑨
Topologies can assume a single player motion class

Evaluation requirement ⑤
Synthetic traces can assume a single player motion class

3.6.3 Motion flow

So far, we have looked at the causes for topology churn in a limited way. The online activity of players and the density in which they gather all have an effect on which P2P overlays networks are the most optimal. Topology churn is however not necessarily caused by player motion, but more superficially *relative* player motion. If all players suddenly decided to move in the same direction at the same speed, the topology would have no reason to change, as each player's AOI remains unchanged.

In NVEs, a case can be made for parties of players that move as a group, or areas of a map that control the flow of motion (e.g. a corridor) in a way that players tend to move in the same direction. If situations like these are common, then topology churn will certainly be affected by this. If we can understand to what extent this is prevalent, it will inform our evaluation requirements as synthetic traces will generally not consider the flow of motion over time. Realistic synthetic traces will not have randomly distributed motion, so it is important that we understand this both for topology design as well as creating synthetic traces for evaluation. Rather, it will have areas with uniform motion in one direction (such as on a road) or where motion in certain directions is more likely than other directions.

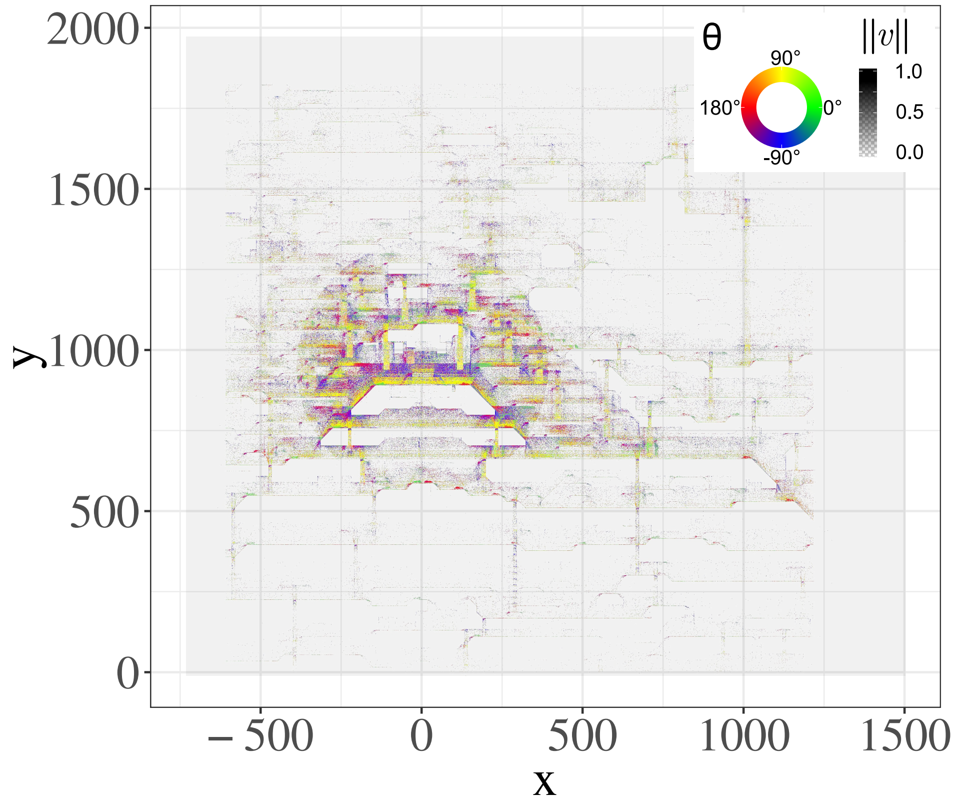
How prevalent is uniform motion across areas?

Figures 3.13 and 3.14 are equivalent to the heatmaps previously (figures 3.7 and 3.8). Every time an avatar in a cell moves, we calculate their velocity based on how far they moved and in how many milliseconds. A cell's final velocity is the mean of all the velocities of recorded motion that has originated from it.

To visualise this information, we took inspiration from "bump maps" (aka normal maps) in computer graphics, which encode a per-pixel normal vector into a texture, where the x, y, and z coordinates are mapped to red, green, and blue channels. The information in these textures is then used to simulate lighting.

Similarly, we take the per-cell velocity vectors, and map the angle of the motion, θ to a range of colours. Up, left, down, and right correspond to yellow, red, blue, and green respectively. Angles in between correspond to intermediate colours on that colour wheel.

We also take the magnitude of the velocity vectors (the speed of the motions), $\|v\|$, and map them to the alpha channel of our velocity map. This makes slower motion more transparent and faster motion more opaque.



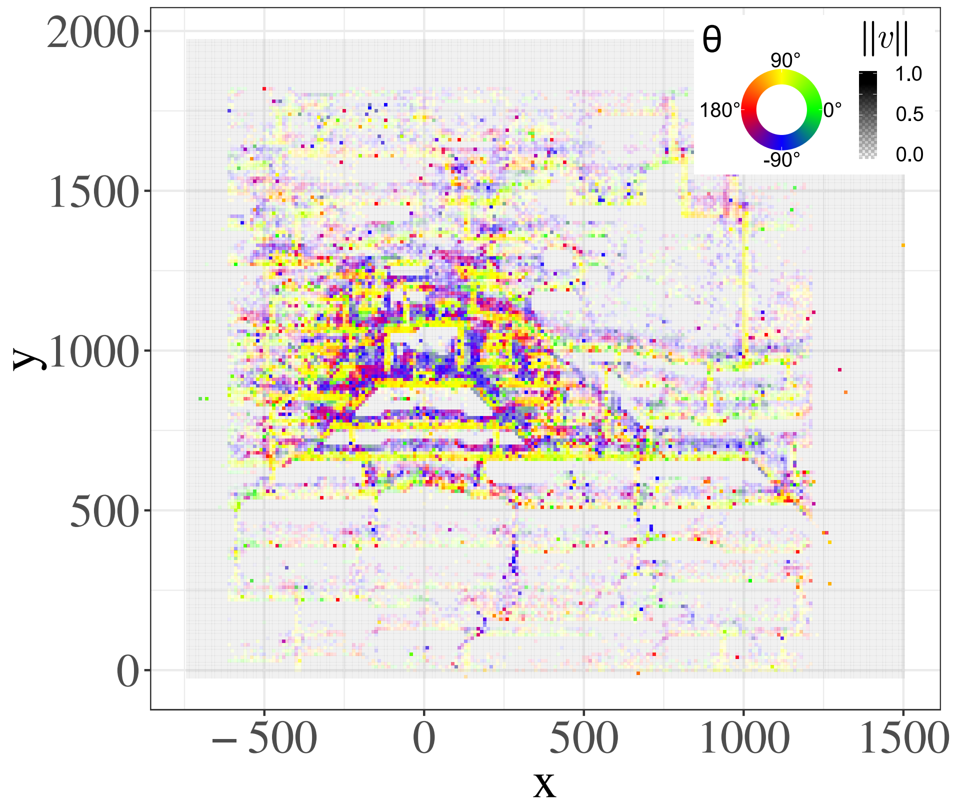
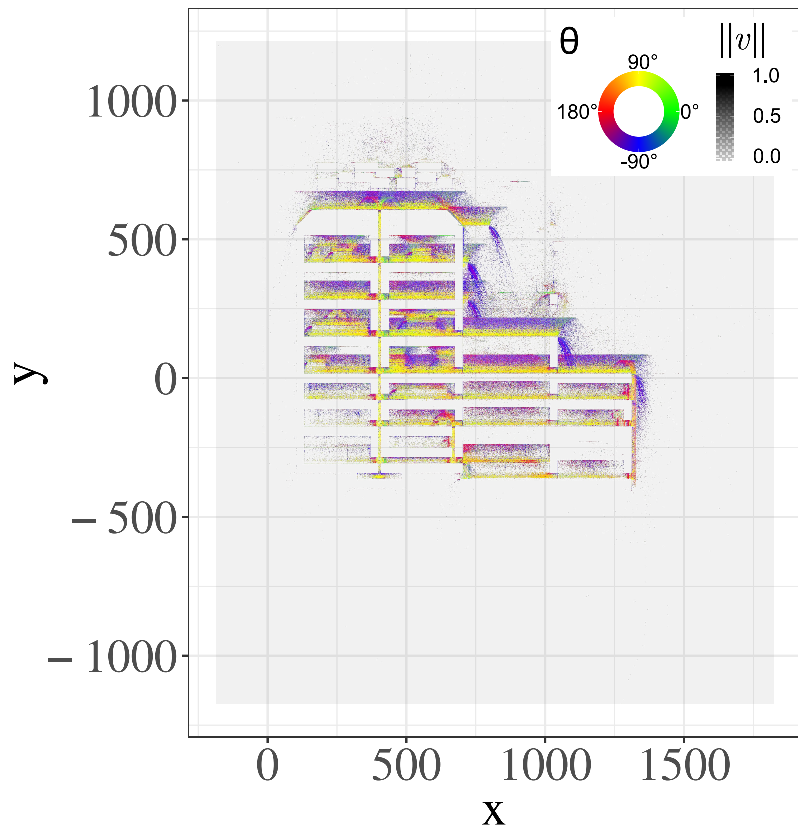


Figure 3.13: Velocity map of the dynamic area with cell sizes of 1×1 (top) and 10×10 (bottom) where opacity is mapped to speed and colour is mapped to angle



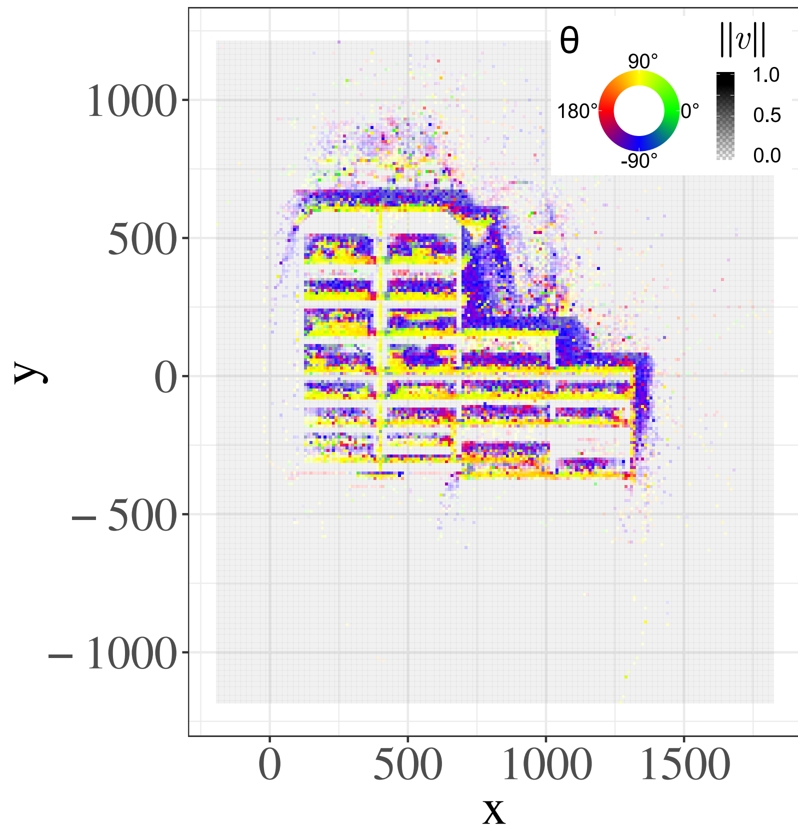


Figure 3.14: Velocity map of the static area with cell sizes of 1×1 (top) and 10×10 (bottom) where opacity is mapped to speed and colour is mapped to angle

In the case of the dynamic area, uniform flows (biased in certain directions) are quite visible in figure 3.13. As players that spawn at the centre tend to move away from the centre, paths that lead away tend to correspond to their colour more strongly. For example, paths that lead up, such as ladders, are predominantly yellow and so are areas where going up is your only choice, such as when there are solid blocks below you. Similarly, where avatars are likely to fall due to gravity, the motion is blue.

In contrast, the flow of motion in the static area flows away from the spawn point of new players (and players not yet logged in) at approximately $(800, -750)$, up through what is a central staircase and into various rooms (the rectangular shapes). This is why the entrance to rooms on the right are green and those to rooms on the left are red. Similarly, there are arches at the top of the staircases at $(800, 250)$ and $(1300, 0)$ that correspond to the divergence of players left and right. The biggest difference however is that the motion is much more uniformly spread in a closed space like this as opposed to an open space like the dynamic area.

Recall that we scale the heats in the occupancy heatmaps logarithmically. Where o is our raw occupancy, we transform this through $1 - e^{-ko}$. Here too we adjust k to make these weights visible and bring the means to a similar order of magnitude, while ensuring that they are the same for the two versions each cell size, such that they can be accurately compared. We do not remove any outliers, as there is a natural limit to the velocities at which avatars can move, as defined by the games mechanics. This area does not have any mechanisms for teleporting or moving unusually quickly. In rare cases, a malicious player can spoof their position, however this was not detected in the week on which we ran our measurements. Table 3.4 summarises these parameters.

Table 3.4: Additional velocity map information

Area	**Cell size**	**k**	**Heat mean**
Dynamic	1×1	10000	0.06910
	10×10	100	0.12630
Static	1×1	10000	0.08017
	10×10	100	0.12640

As we have shown, the answer to our question on the existence of uniform motion flow is clearly yes. Synthetic traces that therefore incorporate a notion of movement between waypoints (see §2.5.4 for more details) are a must. This leads us to draw out an additional evaluation requirement:

Evaluation requirement ⑥
Performance must be measured under workloads with some uniform motion flow

3.6.4 Crowd behaviour

We have not completely covered the motion patterns that could have an effect on topology churn however. We have looked at the global flow of motion, which will tell us how players move at a map level. There is another kind of pattern of motion that can also affect topology churn, which we will examine next.

Earlier we alluded to players clustering in parties/groups and how that can reduce topology churn. If we can show that groups exist in these mobility traces, then this will have implications regarding the workloads we use for evaluation. These workloads must take these crowding patterns into account as they will affect topology churn.

Do players exhibit group clustering?

Now that we have classified avatars as idle and active over time, we can examine avatars clustering into groups. This is common behaviour in VEs and is often ignored by more naive mobility models.

Standard clustering techniques are not well suited to this problem, as group dynamics make for irregular clusters and there can be may group-less individuals and pairs. Fortunately, research into crowd behaviour is a well explored topic, so we can draw from that.

Most techniques do however focus on the social dynamics of real human crowds, where people can face each other and conversations happen within earshot. These are well surveyed, and the most well-known framework is Kendon's F-Formation from the early 90s [82, 81, 25]. In the simplest terms, an F-Formation describes groups that stand around in a circle and how they change. The task of detecting groups is generally considered analogous to detecting F-Formations.

Unfortunately, in-game dynamics are very far from real human dynamics, so we draw on a different, simpler technique from the literature [74] and build on this. We begin by determining a distance below which two idle players can be considered *interacting*. Active players are considered traveling and not part of a group.

Figure 3.15 intuitively visualises why having this threshold is necessary — the AOI

extends beyond the viewport, and players generally do not chat with players at the edge of the screen (especially since their text would be cut off by the viewport). While viewport sizes vary, the game is scaled such that each player can generally see the same distance from where they are located.

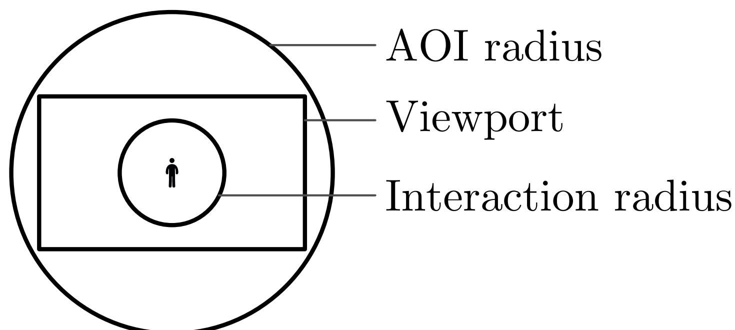
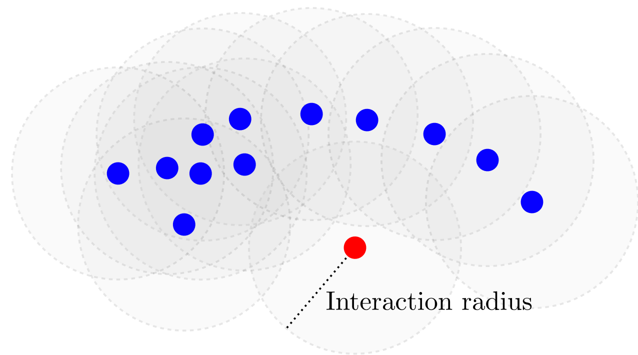


Figure 3.15: A visualisation of AOI radius, viewport, and interaction radius in relation to an avatar and each other

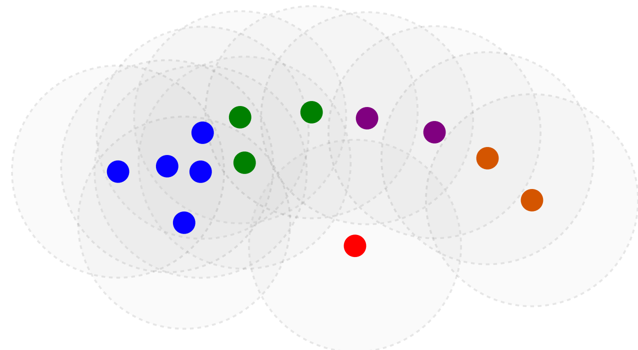
We set the interacting threshold to a third of the AOI. We arrived at this radius through measuring the distance idle pairs of avatars stand relative to each other while speaking in close temporal proximity (to determine if they are chatting or not). To make sure that these pairs are actually chatting with each other, we manually reviewed and labelled the chat logs of 50 such interactions.

Going by what [74] call group formation detection, we can classify avatars into groups with only position data. There is a danger in using this technique however, namely that it relies on the assumption that the number of people in an area remains constant over time, and the thresholds are tuned to this. We have already shown that our environments have highly variable AOI densities however, and the interacting radius must remain constant. This is in spite of the fact that players tend to give each other some space while chatting, as otherwise their chat text (which appears above their avatar) would overlap and become unreadable. We must therefore extend these techniques to take account of this additional challenge.

Group formation detection [74]



Naive greedy clustering



Incremental merging

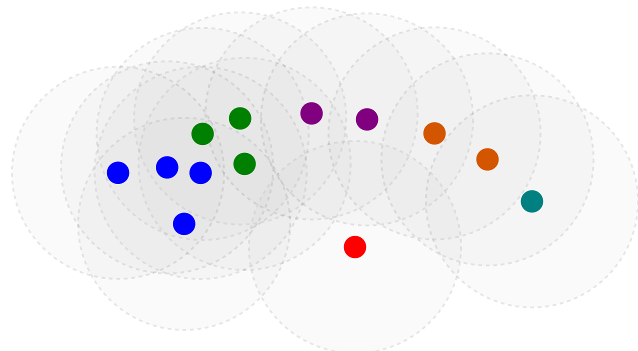


Figure 3.16: A visualisation of avatar grouping as computed through three separate techniques

Figure 3.16 visualises how different grouping techniques would apply to a set of players where each group is represented by a distinct node colour. Active players are already filtered out of these sets so these are all idle and capable of interaction. Idle players with no interactions are considered groups of one. For the sake of demonstration, iteration is done in order of x position (left to right), though in reality we do so in order of player ID.

The first segmentation is arrived at through a simple recursive graph traversal. We recursively visit the set of peers in a players interaction radius and count these. We do this until all players have been visited and the result is a set of groups with no interaction between them.

We can already see the problem with this in figure 3.16, namely that if avatars are too dense, we can end up with oversizes groups. Similarly, if we have a chain of players where each is in interacting range of the previous and the next, then this chain can span the entire map and form a “group”. Clearly these cannot be actual groups as they are not all interacting with each other, and also group conversation tends have an natural upper limit in participant size, or else they fragment and form smaller groups with separate threads of conversation.

The second approach is similar, but we impose an additional limit. During the traversal, any new additions to the group must be within the intersection of all interaction radii of the players within that group. This will result in groups where all members are within interacting range of all other members, which is much more realistic and results in additional, smaller groups.

The problem here is that groups can vary wildly based on which nodes we start traversing from. Because it is a greedy algorithm, early traversals will try to group as many nodes as possible, which can result in lopsided groupings (e.g. blue versus green), or less than ideal pairs at the micro scale (e.g. the rightmost green node and the leftmost purple node are closer together than members of their actual groups).

We have therefore developed a third approach which results in groupings that are much closer to reality. We begin by creating as many groups-of-one as there are peers. Each group has a centroid, which is simply the mean positions of all the peers it contains. At the beginning, this centroid is equal to the only peer each group contains. Then we iterate through the groups, incrementally merging a group with the closest group (by centroid) that satisfies the condition where all members of the of the new merged group must be within interaction range of each other. If there are no more groups that satisfy this condition, we mark the group as stable. We make as many passes over the groups as it takes until all have been marked stable.

We can immediately see the benefits of our grouping method in figure 3.16. The blue and green groups are no longer lopsided and groups are much more contained as a result of the proximity-based merge. The chain is broken up into much more likely and realistic interaction pairs. Overall, our method will result in a higher number of smaller, more contained groups, which is exactly what we want as it matches what we see in reality. This technique is of course just a heuristic, and iteration order will affect the outcome (we iterate by player ID), but it is sufficient for our case. Truly optimal grouping is naturally NP-hard.

The main takeaway from this is that our method solved the most prominent issue, which is that groups may span outside the viewport, for example if a chain of avatars,

each within interaction range of the two neighbours adjacent to them, stand in a row that extends beyond the screen. This can result in unrealistic supergroups, which our method breaks up. Our method can also theoretically produce more accurate groups if we augment it with players' social network information, which our dataset also provides, but this is a direction we have yet to explore.

Figure 3.17 is a plot of the number of groups over time across environments, sampled at 10 minute intervals.

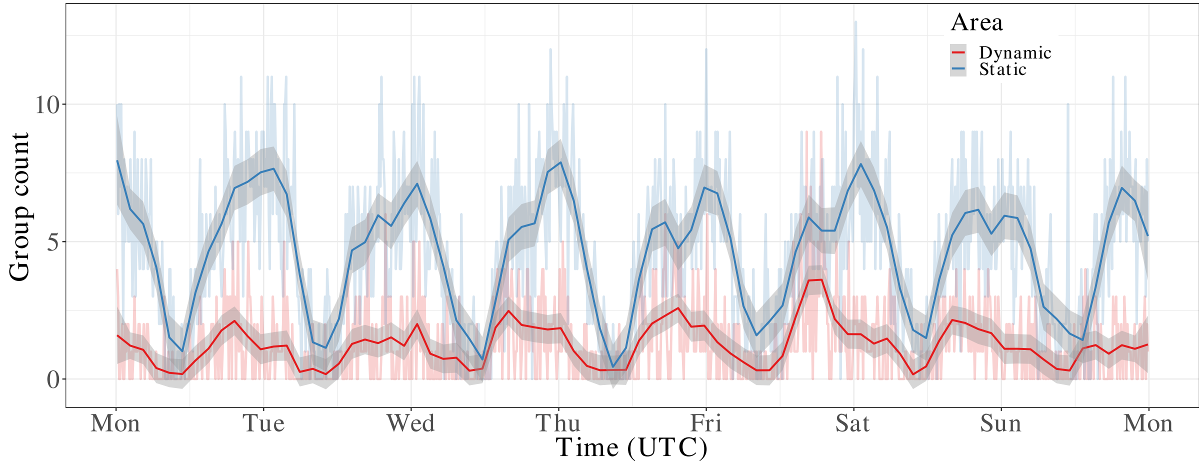


Figure 3.17: A distribution of the number of groups over time across environments

The trend lines, with 0.95 confidence intervals, are computed through LOESS. We see a clear correlation between the number of groups and the number of players online, including diurnal seasonality and continental bimodality. Table 3.5 summarises the distributions of the number of groups over time for both areas.

Table 3.5: Summary of distributions of the number of groups over time across areas

Area	**Min**	**1st quartile**	**Median**	**Mean**	**3rd quartile**	**Max**
Dynamic	0.000	0.000	1.000	1.233	2.000	9.000
Static	0.000	2.500	4.000	4.568	6.000	13.000

At the times with least activity, the players tend to cluster into very few groups (down to a single group). As expected, when many players are online, groups grow in size and split up, akin to social gatherings in real life. Due to the way we detect these groups, and the fact that the dynamic area extends infinitely in all directions, we can rule out the misclassification of groups due to players being densely packed.

We would also like to draw attention to the fact that the dynamic area has very few groups across the board, especially when compared to the static area. Recall that the

dynamic area saw significantly more distinct/unique players over the week; 13,153 to the static area's 4,275. These players are more likely to be in motion however. As a result of this, the dynamic area only has a limited number of groups from the small proportion of players that do interact in some more sedentary hotspots.

The static area is also limited in size, so there is a natural limit to how many groups can co-exist there. If it gets too crowded (although we have not observed this) then players are more likely to leave than crowd closer together. As touched on earlier, this is because players tend to space out while interacting such that their chat text does not overlap with each other and become unreadable.

More interestingly however are the nature of the sizes of these groups. Figure 3.18, which shows the distributions of group size across the week for each area.

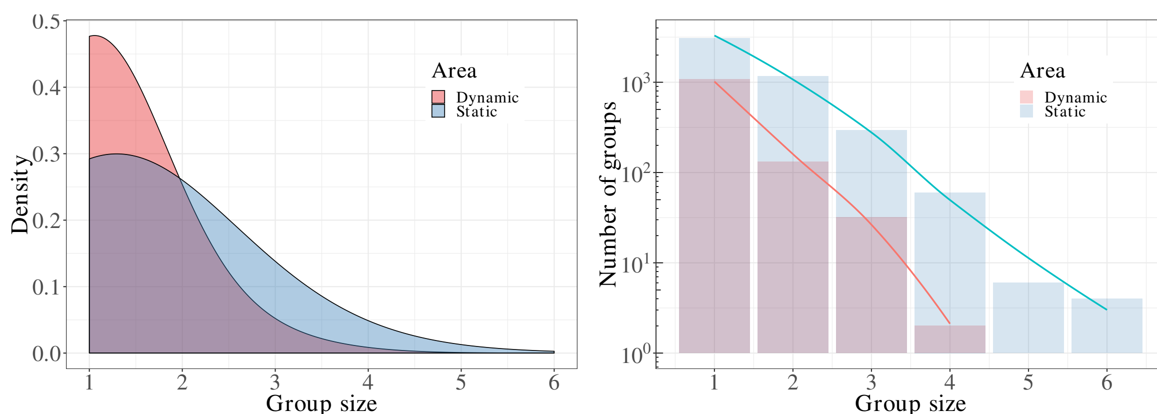


Figure 3.18: Distributions (left) and histograms (right) of groups sizes over time across areas

Here we see clear Gaussian distributions with a high degree of overlap. This is because regardless of the number of total players, the majority are most comfortable in regular-sized groups. Indeed, interaction in larger groups are difficult for the reasons we have discussed so far.

The mean/median group sizes for the dynamic and static distributions are 2.5 and 3.5, with maxima of 4 and 6 respectively. The slight divergence can be explained by the size of the areas — the static area is a limited building, while the dynamic area is literally infinite. In the static area, players cannot simply wander off and explore, so they are more likely to interact with existing groups, though these interactions might not last long.

We included a histogram of group sizes observed in figure 3.18 (right) merely to make two points. The first point is to reiterate that we observed almost four times as many total groups in the static area as in the dynamic area; 4,600 to the static area's 1242 (at our

sampling rate of every ten minutes). The density plot on its own can mislead due to this. The second point is that group sizes follow a roughly similar logarithmic distributions, with the exception of the static area's larger group outliers caused by limited space in the area. These larger groups are observed only fleetingly.

All of this has some benefit to P2P topologies as players do exhibit crowd behaviour and create ephemeral hotspots as a result. For example, connections to peers in the same group can be prioritised versus switching in favour of a peer that is just passing by. Our results show that it is very limited however, since these groups tend to be small, and this therefore does not have a churn reduction effect to the same extent.

Design requirement (10)
Topologies cannot rely on crowd behaviour for churn reduction

3.7 Measuring cheating behaviour

In §2.5.2, we discussed what cheating is in the context of NVEs and the major forms of cheating. We also discussed the kinds of cheating that P2P networks are more prone to, and these usually stem from the lack of an authoritative server. In this section we look at what kinds of cheating are possible, and connect this to the challenges in mitigating these in P2P networks.

In order to determine what types of cheating P2P need to be resilient against, we look at cheating in the context of our dataset. We also look to the Manyland community and, with the blessing of the Manyland developers, have implemented our own set of tools for cheating as a proof of concept [4].

In the process, we found a number of critical vulnerabilities (mainly Cross-Site Scripting (XSS)) which the developers have patched after we have disclosed them. For example, it is possible to paste links to images on the popular image hosting website Imgur, and they are displayed in-game. Clicking these images opens the link in a new tab, however the URLs were not validated and sanitised properly, allowing some special characters that were not escaped to inject arbitrary code into the `onclick` handler. A simple proof of concept is pasting the following link, which the game mistakes as an Imgur link:

```
http://imgur.com/');alert('hi')//
```

[4] `https://github.com/yousefamar/manyland-utils` — This repository is set to private due to the high possibility of abuse. For research access with the Manyland developers' permission, please contact us directly.

This will cause the browser to open an alert dialog with "hi" as if Manyland had sent it. To get around the length limit, code can be injected that loads an external script and runs in in the context of the game. Other characters that get escaped (such as "`.`") can be simply converted from character codes in Javascript or worked around, for example:

```
http://imgur.com/');void($['getScript']('https://amar'+
String['fromCharCode'](46)+'io/evilpayload'))//
```

Since we are running arbitrary code in the context of the game, we literally have full control of the victim's account. This means we can simply make off with their session key and hijack their account, or get more creative by for example:

- Make server requests as the player, instantly deleting all their belongings, areas, and creations (akin to ransomware)
- Ban the player's account by triggering a cheat detection automatic ban
- Create official-looking dialogs and phish information
- Wait until the player attempts an in-game purchase, then replace the PayPal button with a link to a phishing site to steal PayPal credentials
- Switch the player over to a makeshift private server
- Stream the game canvas, or tee all their outbound traffic, to somewhere and spy on them
- Escalate by triggering a download of a Chrome extension (or even an actual key-logger) and pretend it is an official Desktop client or such

Similarly, a server-side bug caused the IDs of items attached to forum posts to not be properly validated, and where an `onclick` handler would collect that item, we could again inject arbitrary Javascript. Besides these random XSS vulnerabilities, we discovered a range of denial-of-service style vulnerabilities. For example spoofing your velocity to an impossibly high number (or indeed spoofing any number of state variables) would completely freeze the client as it tries to handle the physics simulation. Sending other invalid characters would outright crash the tab (some of these were bugs in Chrome itself that were patched[5]) and spamming very long chat messages would scroll client-side rendering to a crawl.

[5] `https://andrisatteka.blogspot.com/2015/09/a-simple-string-to-crash-google-chrome.html`

It is unfortunately not very difficult to find exploits like these in games if you dig deep enough. These kinds of vulnerabilities are however less relevant to us, as they constitute "hacking" rather than cheating. For the remainder of this section, we will examine the ways in which cheating manifests itself and how it could affect P2P topologies.

What kinds of cheating are common?

Going by Gauthierdickey et al. [47] taxonomy, which we discussed in more detail in §2.5.2, we can ignore game-layer cheats, as these are a caused by game design holes, which are game-specific and irrelevant to us. Similarly, network-layer cheating (denial-of-service etc) are out of scope for the reasons we discussed in §2.5.2. Unsporting behaviour like disconnecting right before losing is of course also prevalent here, but we do not focus on this kind of "cheating" because it is not a network-level problem and can be discouraged through game design. This leaves us with application-layer and protocol-layer cheating.

It is well-known in the Manyland community that there is very little server-side validation to limit server costs. The servers does not check if a player's state is attainable, and will generally take the player's word for it. This makes protocol-layer cheats unnecessary, because for example, why delay your packets to get a timing advantage when you can just make yourself immortal instead?

The most common manifestation of this is players tweaking client-side gravity in order to reach inaccessible areas as physics simulation is entirely client-side. This is common because all it takes is modifying an easily accessible, unobfuscated variable: `ig.game.gravity`.

The first measure that Manyland takes to combat this kind of cheating is through obfuscating the client source code, and reobfuscating it every time there is an update (such that the variable and function names are changed for example). This can act as a reasonable deterrence to low-skilled cheaters. There are however players that are willing to go through the effort to get past this, and worse, distribute the scripts that they write to lower-skilled players.

A common example of this is players spoofing their own positions. This can be done to get past physical barriers, but more often it is done by players who want to show off that they can "hack". This is less trivial, because the player object is difficult to find/access due to obfuscation. Few cheaters automate this process, instead looking for the variable manually.

The proper measure to preventing this kind of cheating is server-side validation — the

server can ban players who move at impossible speeds for example. Due to server-side limitations, Manyland instead relies on other players to spot this kind of cheating and flag the offending player.

There is a way to detect some instances of this kind of cheating client-side. Recall that we have calculated player velocities over time to analyse player motion. Occurrences of players spoofing their position will show up as a movement with a speed higher than the maximum speed possible as defined by game logic (130 units per second). This of course does not account for small movements through walls, but that can be detected too with knowledge of where walls are.

To detect these, we scan through the velocity traces we derived in §3.6.3 and count the instances where this threshold is exceeded. Within our week's trace, we have detected:

- 45 instances of position spoofing in the dynamic area
- 518 instances of position spoofing in the static area
- 37 out of the 12988 players seen (0.28%) have been caught cheating in the dynamic area
- 234 out of the 4,248 players seen (5.5%) have been caught cheating in the static area

These numbers are actually quite large relatively speaking and indicative of a "scripters" epidemic that the game has been facing. We note also that the static area has significantly more cases of this. This is due to the fact that this social area is unmoderated and cheaters like to show off their scripts here, as well as due to the existence of an unreachable ledge that can only be reached through cheating, which many take as a challenge.

Cheating is even more consequential in smaller areas, especially since, where it matters, cheaters try to hide their actions such that they are not condemned or flagged. Cheating like this is also quite bursty — one player will spoof their position one or more times in a short period of time and stop once their goal is fulfilled or they are banned.

Manyland developers have since implemented server-side checks for players who attempt to spoof critical properties such as their rank or their state as an area editor. This is quite effective, since most cheaters will not bother to find out that this check relies on a client update that can be disabled, making these state changes invisible to the servers but still able to influence client-side logic (such as unlocking doors that only open to area moderators).

Historically, players who send malformed updates can freeze and crash any client that attempts to parse these. This has since been patched through client-side validation. With the permission of Manyland developers, we attempted to find any way possible to cheat by using the taxonomies from the literature as a guide to aim at a wide coverage of all cheating vectors. Other existing cheating mechanisms that we have confirmed through our proof-of-concept extension include:

- A range of movement cheats, including disabling gravity, changing movement speed, or teleporting just by clicking the mouse cursor at a specific spot under a hotkey.
- Immortality, disabling fall damage, immunity to the effects of any items
- Spoofing client-side moderator status to access locked areas and use restricted items
- Suppressing client-side cheat detection to circumvent automatic bans
- Panning and zooming the game camera (cf "maphacks" in §2.5.2)
- Automation of slow game interactions such as building or removing structures
- Inspecting and overriding the behaviour of objects with client-side logic
- Applying the effect worn equipment to the player without having that equipment
- Inspecting hidden player stats and cloning player skins and items
- Collecting items en masse
- Blocking ads (creates more screen real estate; cf "maphacks")
- Spamming visual effects such as particles

The reason we list this is to point out that literally all of these can be solved through server-side through various means, many of which we have already discussed in §2.5.2. However, server-side validation is expensive both in terms of server costs and development costs. If we bring over the intents of application-layer cheaters to the P2P world, we can see that these issues can instead be solved through client-side validation. For example, if your client sees a player flying through walls, it can immediately know that they are cheating. It is the game developers responsibility/decision to put this validation in place either server-side (in client-server architectures) or client-side (in P2P architectures).

The problem is what comes next? How can non-authoritative peers collectively decide that one or more of them are cheating and should be kicked out of the network? This is a

difficult problem, and we already touched on consensus voting and their problems (Sybil attacks etc) in §2.5.2. These techniques are many, and far outside of the scope of this thesis. We are however interested in indirect cheating mitigation at the network level. In other words, we assume that there is some system in place that can automatically or manually classify players as disreputable (for example, in Manyland, there is a vote-based reputation system which we discuss in the next section) and we can simply use that information to augment neighbour selection, such that traffic is less likely to flow through cheaters.

We have also proven that it is possible to spoof persistent state (currency, inventory, etc). There are many interesting approaches to mitigating this (many game-layer solutions on which node should be an authority on something exist), however, as discussed in §2.5.2, this is unfortunately out of scope of this thesis, because it is unrelated to UD. In fact, this can be completely centralised, especially given that this class of traffic is virtually negligible in volume compared to UD traffic, so there is much less incentive to decentralise the propagation of persistent state in the first place.

There is one substantial challenge that exists for UD in P2P networks (specifically ones where peers may forward the communication of other peers) but not for client-server networks. We need to ensure the integrity of messages in transit. This is not necessarily a topological requirement but it is still an important design requirement.

Finally, while we have not observed protocol-level cheating as it is not possible to do over a client-server network, it is common in other NVEs [47]. Assuming the integrity of messages is maintained, the literature indicates that players gain an unfair advantage by dropping or delaying packets. Some protocol-level cheating will always be a problem regardless of the architecture (e.g. colluding with other players [47]) but these timing attacks are more pronounced in a P2P context where peers must forward the packets of other peers too (not just sending their own). A serious system cannot get away without mitigating this by design. This discourse on the credible forms of cheating leaves us with two additional requirements:

Design requirement (11)
The integrity of messages in transit must be guaranteed

Design requirement (12)
Network must be resilient to deliberate dropping/delaying of forwarded packets

3.8 Measuring player reputation

We have previously alluded to the existence of a reputation system in Manyland, which is one of the attractive features of using our dataset for research. Players have a global rank between 1 and 5, where 5 (also known as "ring rank" in-game) is the most reputable and gives players higher moderation privileges. These mainly include the ability to edit global spawn areas, place "dangerous" objects in public areas (less reputable players may abuse this privilege), and boot players with new accounts out of public areas. The developers of the game have a rank of 10 (which on the client-side gives their players access to diagnostic and debug tools), and players that have been banned have a rank of 0.

It is not unprecedented that P2P networks use social metrics for neighbour selection [56]. The purpose of this is twofold, both for improving topology performance. Firstly, players are more likely to seek out and interact with reputable friends in their AOIs. Secondly, disreputable players (who may engage in cheating) can have a lower priority in neighbour selection, which creates networks where fewer paths go through cheaters.

We already know that reputation can be used to enhance P2P neighbour selection, but we do not know if vote-based rank can. In order to also evaluate reputation-based neighbour selection, we need an alternative proxy for reputation in order to set the likelihood of a player cheating in a simulated/emulated network. It is an open question how much reputation corresponds with vote-based rank. We ask:

Can vote-based ranks / social networks be used as a proxy for "reputation" to be used in reputation-based P2P neighbour selection? Can other metrics (e.g. account age) for evaluation purposes?

This section is split into two subsections. The first focuses on reputation via vote-based rank, and the second via social network metrics.

3.8.1 Vote-based rank

We took a sample of 223,040 players by querying the profiles of player IDs we have sourced through various means, such as the players our bots have seen, the creators of items our bots have seen, authors of crawled forum posts, and several other smaller sources. These profiles are behind an undocumented Representational State Transfer (REST) API that takes a player ID as a parameter and returns profiles as JSON objects. Once we found the correct endpoint, we queried these profiles while also waiting a second between requests so as to not put undue load on the game servers. It took some days to query all these

profiles in this way, but once we had them, we only update them at much less frequent intervals. These collected profiles contain a lot of useful player metadata, including player rank. We also keep track of changes in rank, for as long as our bots are running and observe a change.

Figure 3.19 shows how rank is distributed across the players in our dataset.

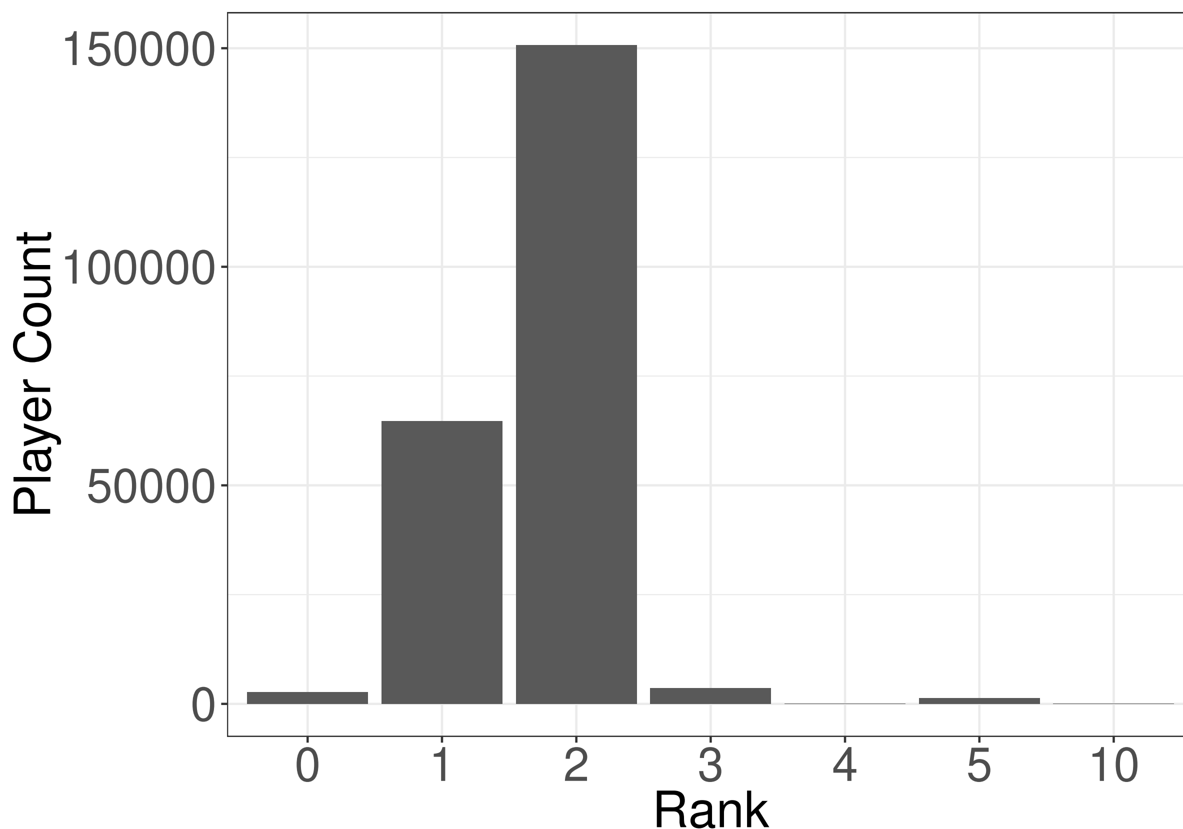


Figure 3.19: A histogram of player ranks

As expected, ranks are mostly middling, with a higher number of rank 5s due to that being the maximum ranks and players increasing in rank over time. The number of developers never changes and they are very few. Rank 4 is also largely empty, as it acts as a kind of temporary queue for players that have already reached the criteria for rank 5. Developers manually check this queue and promote players to rank 5 in fixed batches periodically. Most players do not go over rank 2 as they either do not play long enough, play very casually (not utilising the full features of the game), or do not interact with high-level players whose votes carry a lot of weight.

As we have the account creation dates for these players, we can also plot a histogram of that information to see if it can be used as a reputation proxy at all. Figure 3.20 depicts this.

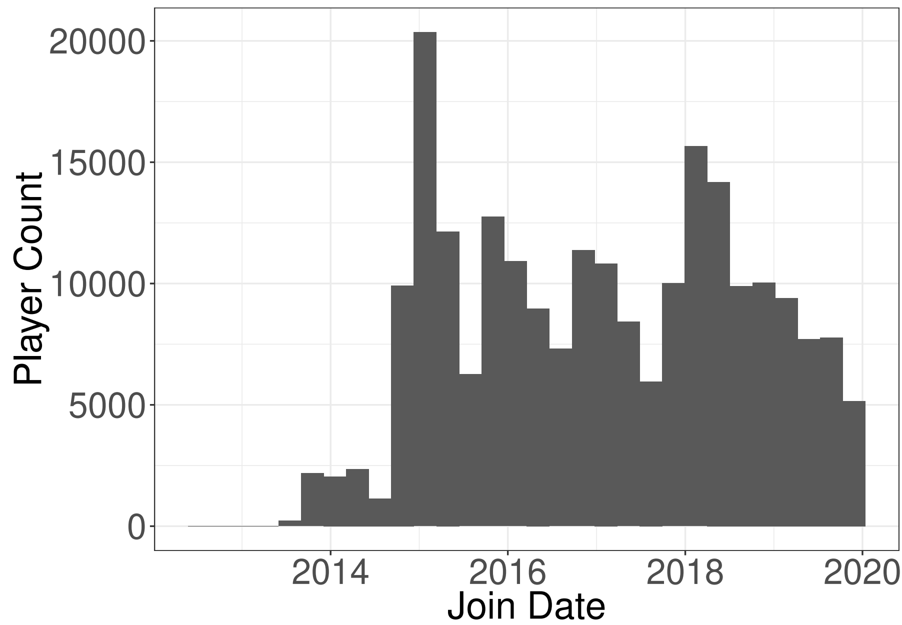


Figure 3.20: A histogram of player account ages

Knowing Manyland's history, we can see that the increases roughly correspond to historical events, such as a viral posts on HackerNews and Reddit, the Chrome Web Store feature, promotion through news publications, and the Steam release. Further, figure 3.21 shows how player rank relates to player account age.

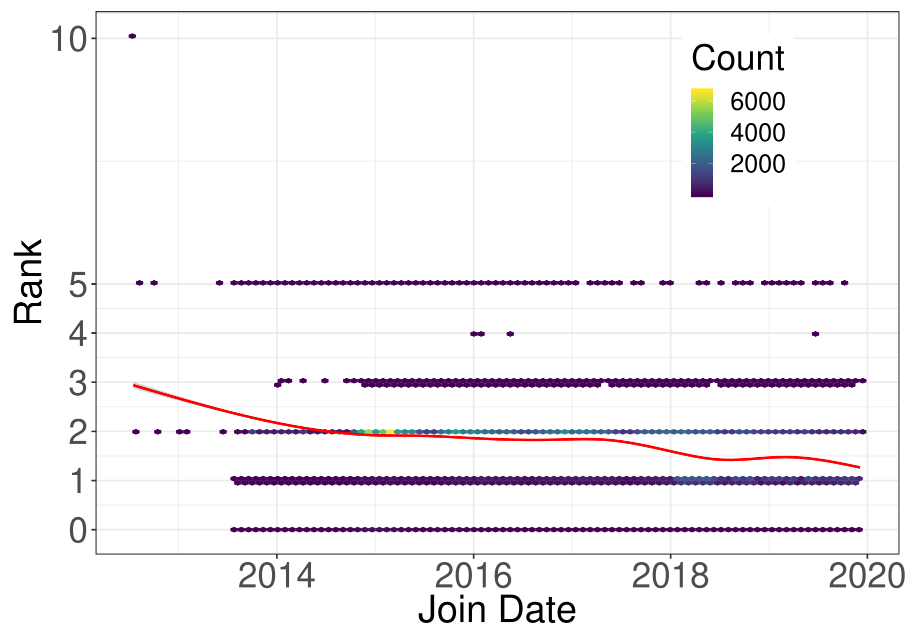


Figure 3.21: A scatterplot of players binned into 100 vertical and horizontal hexagonal bins of player rank against player account age with a Generalized Additive Model trend line (red)

Here we can see a weak positive correlation between account age and rank, which is as a result of how the in-game rank mechanic is designed. As there were far more players in the past, those that came during those spikes and stuck around now largely make up the rank 5 cohort. There is an in-game sentiment that these "ring-ranks" are an oppressive elite from that time, making the barrier to rank 5 seem more amplified than it is. These results may incidentally explain that sentiment. This tells us that age can indeed be used as a broad proxy for reputation but ignores the finer details — reputable new players and disreputable old players are lost noise.

3.8.2 Social network metrics

We alluded to the use of social networks in group detection. The literature also shows that this information can be used for P2P neighbour selection algorithm that mitigate Sybil attacks [92]. This kind of metadata is therefore invaluable to our dataset. We can use much stronger and more continuous social network measures to see if these correlate with rank. With these measurements, we can examine the validity of vote-based rank as a proxy for reputation-based P2P neighbour selection metrics more confidently, as social network metrics provide a more fundamental "ground-truth" for reputation, unlike a vote-based integer rank from one to five.

While Manyland has a friends list system, where friends can find each other and share updates among other functionality, these friends lists are private. On the other hand, players can send each other items with a short message attached in-game that are public by default. These gifts are called "mifts". Sending mifts costs a small amount of money, or is free for subscribers, as a means of monetising the game. Mifts serve no purpose except as a gesture of social accolades.

Similar to how we crawled the 223,040 player profiles, there is another endpoint that lists public mifts page by page. We query all of these in the same way and append that information (who sent them and when they were received) to our database of player profiles. Incidentally, one of our sources of player IDs was by recursively visiting mifters, the players that have mifted those, and so on. The mift crawler discovered only a single new/unknown player during this process who sent a mift, which is a testament to the coverage our other collection methods.

This data creates a much stronger measure of social networks, as the number of mifts one person has given another indicates the *strength* of a social bond, as opposed to the binary nature of a friends list. The number of mifts a player has received in total is a good indicator of their social standing, especially if they have received mifts from high-

ranking players or developers. The mift network is directed; asymmetric mift-giving relationships can exist, most notably as players give the main developer many mifts, but not necessarily the other way around. While the monetary barrier to sending mifts excludes e.g. the younger demographic from participating, the data on the participant is much more precise. In order to capture the excluded demographic, this data can in theory be augmented with measures such as the time users spend in the same group.

Of the 223,040 players we sampled, only 2,866 (1.28%) were involved in public mift-related activity (giving/receiving). This is understandable considering the monetary barrier. Even considering just a subset of these however create social network graphs that exhibit the same patterns and characteristics.

Unlimited mifts for subscribers (aka those with "minfinity") are underutilised — at the time of writing this, there are solely 23 players with minfinity, which even in our limited sample account for 0.01% of players. As such, the monetary barrier to expressing social praise in the form of mifts is generally the same across all players. It is typical in most games that a small minority of players spend the most on in-game purchases, and unscrupulous studios in fact target and rely on these "whales" to drive their profits.

We take this information and build a directed graph with weighted edges, where nodes correspond to players, edges to mift-giving, and edge weights to the number of mifts given (ever). We also set the node radii to the weighted in-degree of the node, which can roughly correspond to the "popularity" of a player, as it is the sum of mifts they have received.

We then lay this graph out using the OpenOrd [96] force-directed layout algorithm. This is based on the much older and more well-known Fruchterman-Reingold algorithm [40] with simulation phases that make it generally more well suited to visualising large graphs. Unlike other algorithms, it "pulls out" clusters, which is ideal for social network and other small-world network graph analysis. Bear in mind however that edge lengths correspond less to edge weights than with other algorithms, so what looks like a satellite community is not necessarily one.

We rendered four different versions of this graph to better illustrate the different lenses that we can examine this data through. In all versions, edge thicknesses and darkness correspond to edge weight.

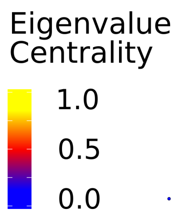


Figure 3.22: Mift network partitioned by eigenvector centrality

In figure 3.22, we begin by colouring the nodes by eigenvector centrality after 100 iterations. This is a well-known measure for node influence in a network. Google's PageRank and Katz centrality [75] are based on this metric.

Here we can immediately see that there are a core cluster of influential nodes, however these interestingly do not correspond one-to-one with the most mifted nodes (the most mifted node is one of the developers). The most influential node is a popular member of the community well known for his artwork.

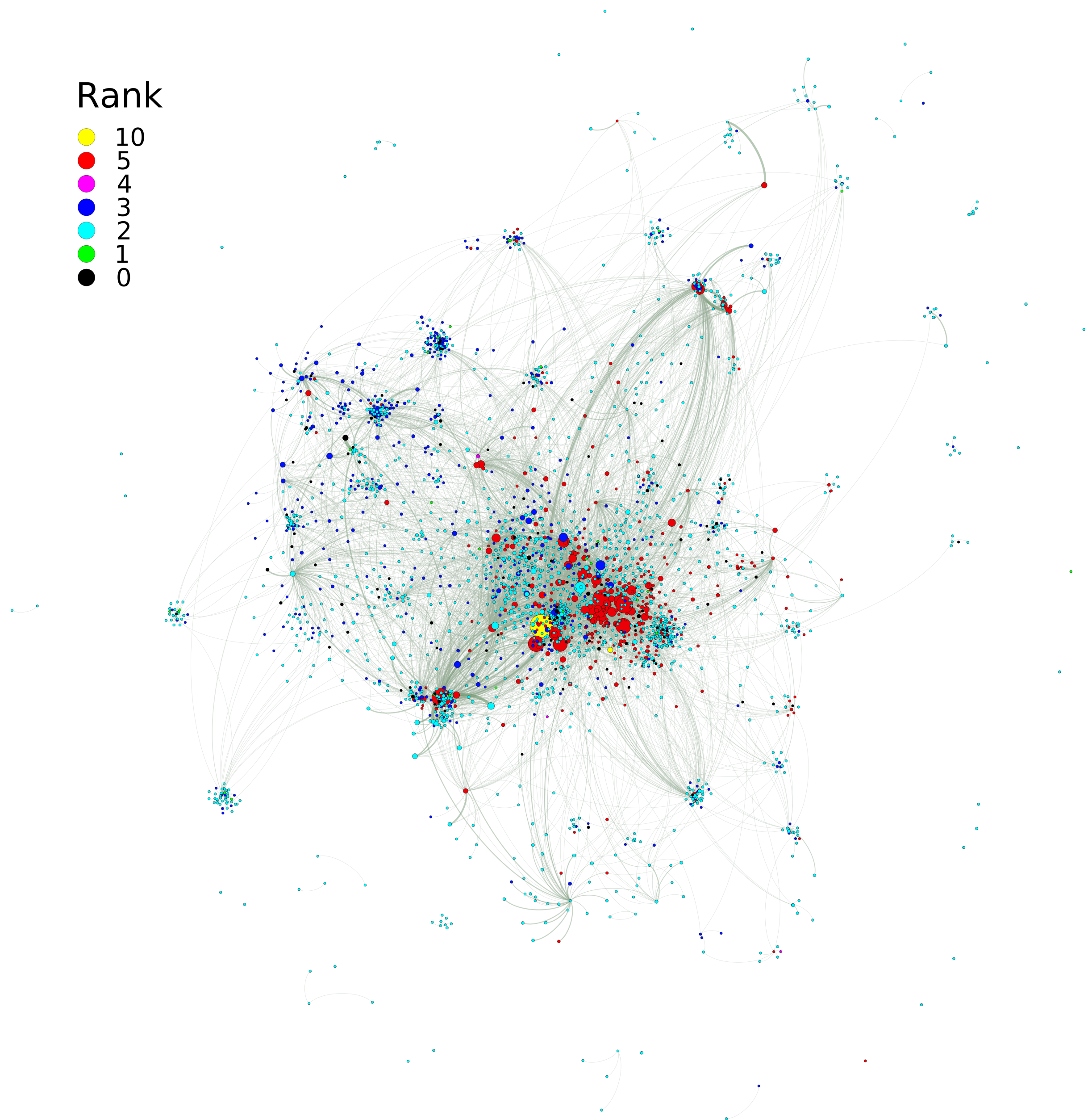


Figure 3.23: Mift network partitioned by player rank

We compare this to a graph coloured by rank (figure 3.23). Note however that the rank distribution across the mift network is not the same as the overall rank distribution across all players. Here, 0.07% of nodes are rank 10 (developers), 15.35% are rank 5, 0.1% are 4, 19.3% are 3, 59.94% are 2, 0.98% are 1, and 4.26% are 0 (banned). The rank distribution here is more skewed towards higher ranks than the overall distribution. This has many reasons, primarily the fact that older and more reputable players (recall the correlation between age and rank) are more likely to make an in-game purchase in the first place. Further, the act of giving a mift is social, and can increase social standing in and of itself.

It is clear to see that players with a rank of 5 occupy more central positions while lower ranks are more dispersed, just as with eigenvector centrality. To better illustrate this relationship, we plot rank against eigenvector centrality in figure 3.24 This shows that player rank can be used as a valid indication of node reputation.

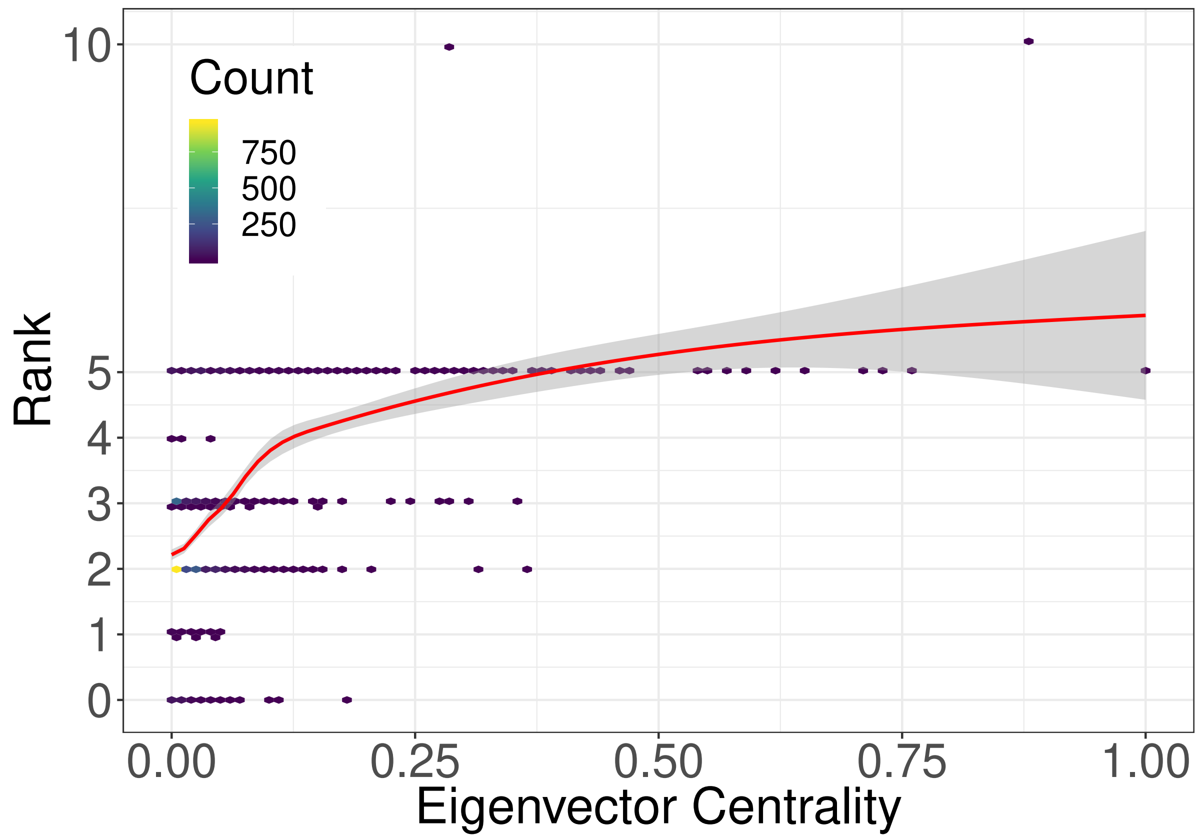


Figure 3.24: A scatterplot of players binned into 100 vertical and horizontal hexagonal bins of player rank against player eigenvector centrality with a Generalized Additive Model trend line (red)

We have previously shown a correlation between player rank and account age. For good measure, we colour nodes by player age in figure 3.25.

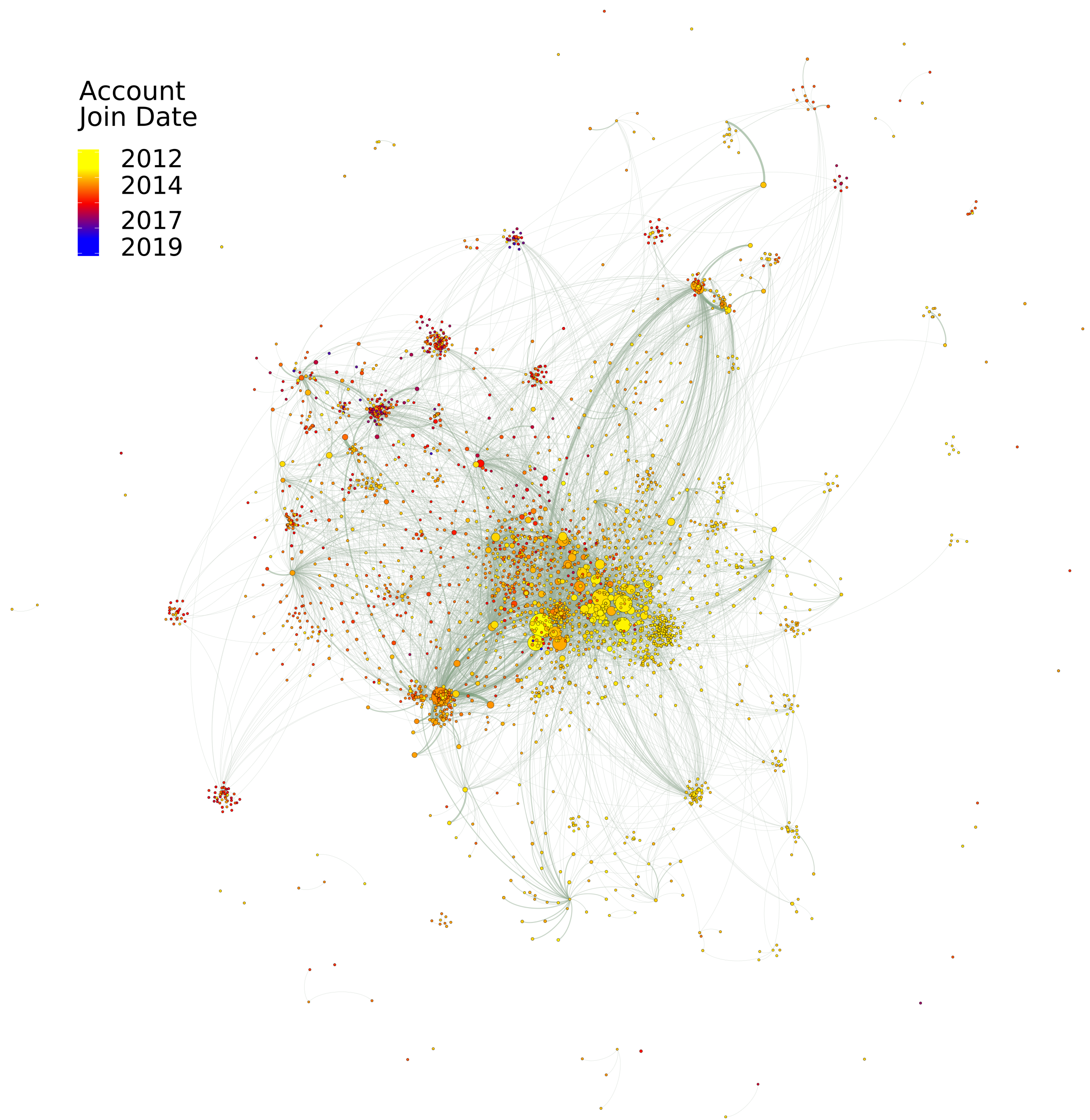


Figure 3.25: Mift network partitioned by age

Account age within the mift network reinforces our earlier observation that age correlates with social status. Note however that, while it is possible to hide received mifts, older players are more likely to have more mifts just by virtue of being around longer and interacting with others. We plot the received mifts (node weighted in-degree) against rank and age in figure 3.26 to illustrate this.

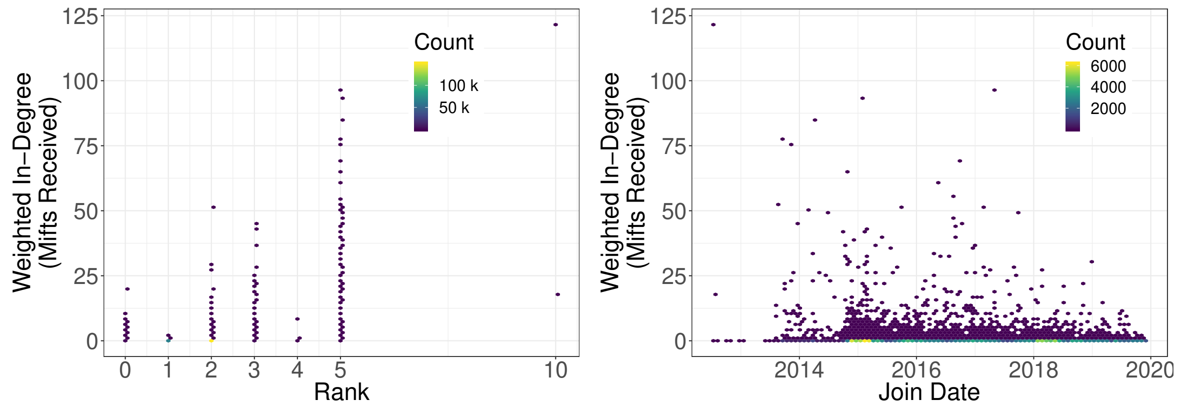


Figure 3.26: Scatterplots of players binned into 100 vertical and horizontal hexagonal bins of player weighted in-degree (mifts received) against rank (left) and age (right)

We can see how, age-wise, the number of players receiving mifts roughly correspond to the numbers of players who joined at those dates (see figure 3.20), with the exception of the developers on the top left (oldest accounts and most mifts received). If there are more players of a certain age, it makes sense that players of that age receive more mifts. Along with the conclusions of the previous subsection, this tells us that age is a weak reputation indicator.

Figure 3.26 (left) on the other hand shows us that we can roughly expect higher-ranked individuals to receive more mifts. That being said, players who were at one point reputable and became disreputable can lose their rank and in extreme cases get banned, hence quite a few rank 0 mift recipients. This is a much stronger indicator of reputation and social standing.

Finally, in figure 3.27 we colour nodes by modularity class. To do this, we use a standard technique [11] with a resolution of 1.0 [89]. The graph unfolds into 51 communities, with a modularity of 0.54 (taking edge weights into account).

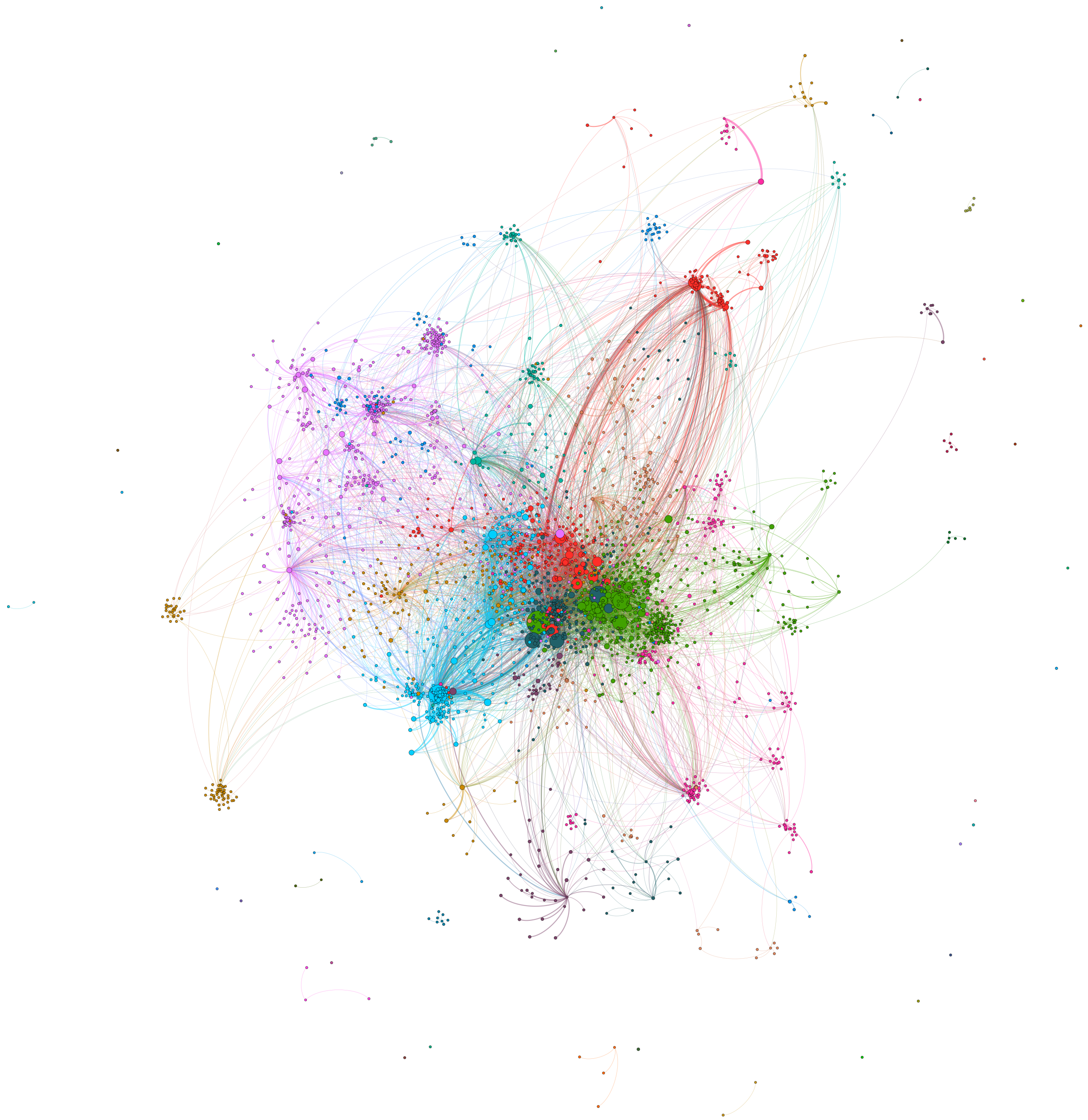


Figure 3.27: Mift network partitioned by community / modularity class

The way we have laid this graph out can make it look like there are clusters of satellite communities which one would expect to have distinct colours, however this is not the case. These clusters are oftentimes as a result of a few players who send a lot of mifts (which also has the effect of devaluing their influence). The communities, which are based on edge weights as opposed to layout position, more accurately reflect the social dynamics within the network.

We can mostly see that players with similar ranks and age tend to form communities that can vary in connectedness with each other. This makes sense as players who

started playing around the same time would have played together. With ground truth geographical data, or better yet, granular information on when players on online, we can additionally investigate if these communities correlate with geography or timezone in any way. Collecting the data to allow this kind of analysis is a future goal. We can also see when a mift was sent, so there is an unexplored temporal aspect to this social behaviour. Similarly, analysis of the purpose of these mifts is also a possibility, as each have a message attached (for example wishing happy birthday).

We note that while this social network exhibits some characteristics of a small-world network, it cannot be considered one. Table 3.6 summarises a range of network statistics.

Table 3.6: Summary of mift network statistics

Mean degree	2.512
Mean weighted degree (mifts received)	3.492
Network diameter	10
Graph density	0.001
Modularity	0.54
Weakly connected components [128]	40
Strongly connected components [128]	2689
Mean clustering coefficient	0.116
Mean path length	3.67

The degree distribution of the network shows a high prevalence of hubs that make the network have an overall smaller diameter. The mean path length is also relatively small, which is typical of a small-world network. However, the average clustering coefficient is too low for this network to contain clear cliques, which is not characteristic of a small-world network.

The data we have analysed in this section can be used either for VE social network analysis purposes, or in augmenting neighbour selection algorithms, which we focus on in this thesis. Since there is no authoritative server in a P2P network to keep track of these metrics however, we must also by design ensure that they are not misrepresented. We draw out from this a design justification and requirement we seek to pursue to augment our solution, as well as an evaluation requirement:

Design requirement (13)
Misrepresentation of neighbourhood trust/reputation metrics must be effectively mitigated

Design requirement ⑭
Neighbourhood selection metrics should incorporate rank as a reputation proxy

Evaluation requirement ⑦
Eigenvector centrality can be used to assign bad actors realistically to evaluate a network's resilience to cheaters

3.9 Measuring device heterogeneity

An important consideration in a P2P context that is less of a question with traditional client-server architectures is that of the heterogeneity across clients/peers. Indeed, in §2.7.1.3, we explored how some superpeer topologies take advantage of resource heterogeneity in superpeer selection. This leads us to ask:

Is device resource heterogeneity significant?

Unfortunately, it is very difficult to answer this question deeply, as it involves collecting user statistics to which we simply do not have access to, especially across games. In this section we therefore only try to establish if it is a factor that must be considered, rather than to exactly what extent.

To aid us in answering this question, Manyland developers have kindly given us insight into their own high-level analytics and given us permission to include these here. We have also included other statistics that reinforce the measurements we have taken throughout this chapter, including activity over time (figure 3.30), by location (figure 3.29), area distribution (figure 3.28), diurnality (figure 3.29) etc. Indeed some of this information can serve as ground truth for inferences we can make without it (e.g. geography based on an inference of timezone from user activity and other data points).

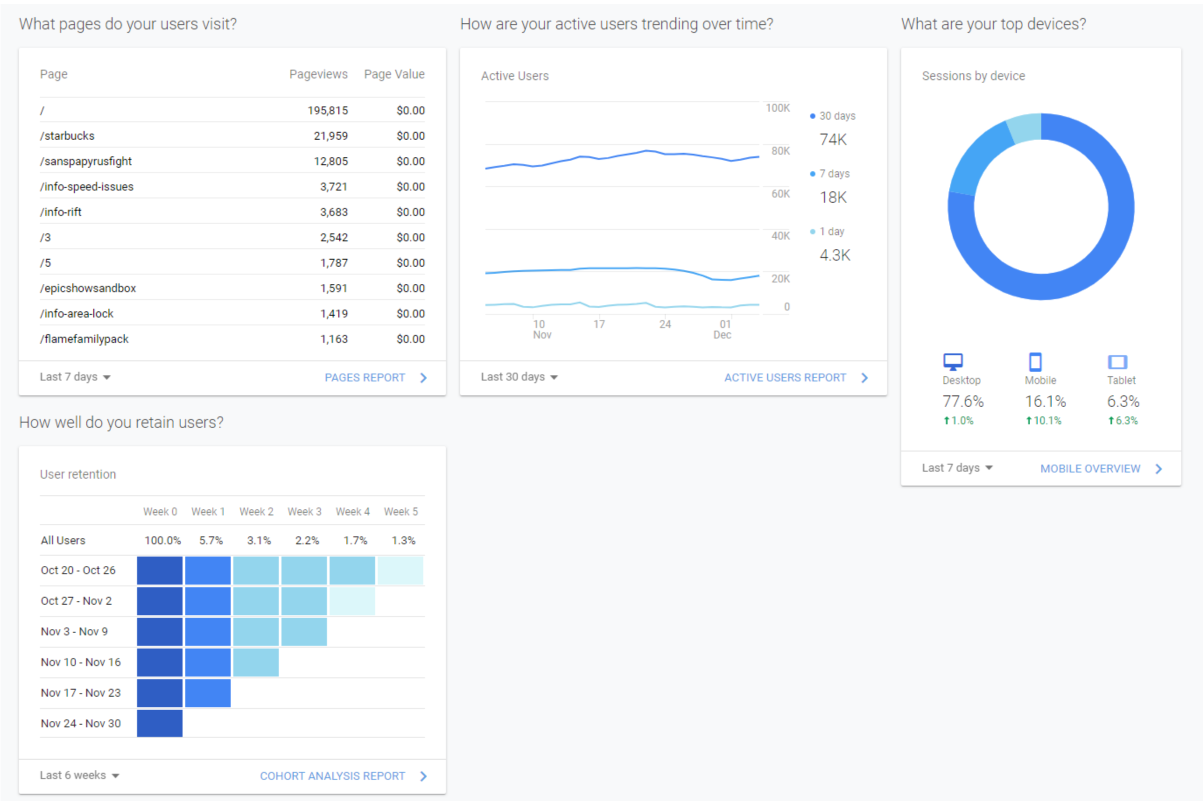


Figure 3.28: Manyland.com Google analytics as of 2019-12-05 on pages, user activity, devices, and retention

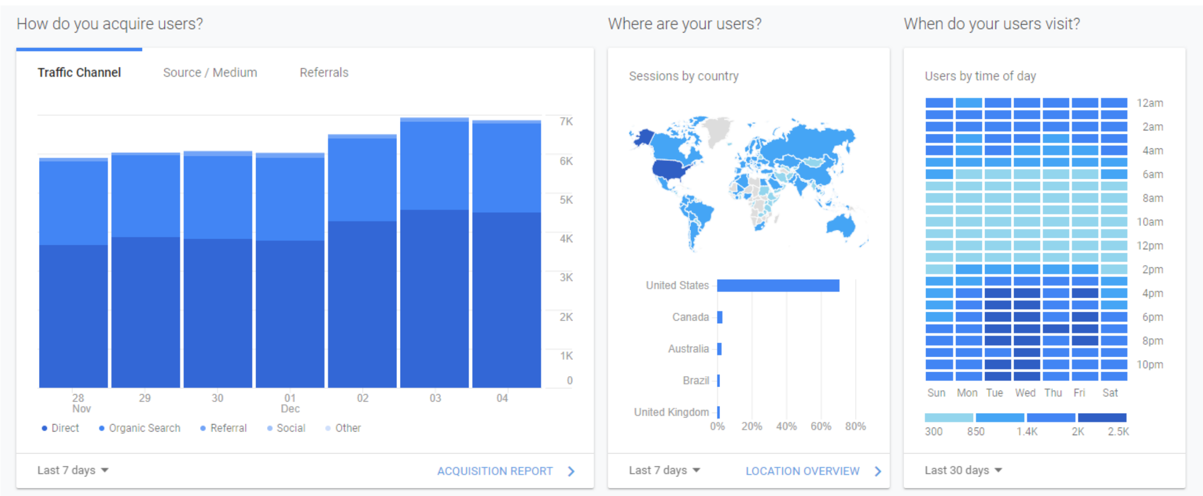


Figure 3.29: Manyland.com Google analytics as of 2019-12-05 on user acquisition, geography, and timing

Figure 3.30: Manyland.com Google analytics as of 2019-12-05 on user sessions

In figure 3.28 specifically, we can see that 16.1% and 6.3% of users use mobile phones and tablets respectively, versus the 77.6% desktop users. This is, of course, not representative of all browser games, but is to be expected of a "desktop-first" game, where the mobile apps are simply an afterthought WebView wrapper over the desktop game.

Note also that while this VE is primarily intended for desktop interaction (56.7k by web clicks), there as still a significant portion of mobile and tablet users (19.9k and 3.64k respectively). This further drives the heterogeneity constraint when designing P2P topologies, not to mention heterogeneous resources within those categories.

There can be of course browser-based games that are desktop-only, and here the constraints may be laxer, however the system we design should be general enough to account for — and take advantage of — this heterogeneity. This information generates three additional design and evaluation requirements:

Design requirement (15)
Topology must account for heterogeneous devices with varying network limitations

Design requirement (16)
Topology must account for heterogeneous pairwise links at the network level due to geography

Evaluation requirement (8)
Evaluation must be over heterogeneous networks

3.10 Measuring browser constraints

As discussed in §2, the only way to currently create P2P connections in the browser is via WebRTC datachannels. This introduces many challenges that would not exist outside the browser. We have already captured one challenge in §3.9, namely that peers and connections are very heterogeneous, as they range from desktops to mobile phones, fiber to 3G. Although WebRTC has been widely supported for years now, the standards have not yet settled, and using these APIs is widely considered a headache by developers due to how complex and fragile they are.

Current APIs do not allow sending more than 64KB of data at a time (any more than this has to be chunked). If a Firefox peer has to send to a Chromium peer, this limit is 16KB for an ordered and reliable datachannel. We do not consider this a legitimate challenge, as updates in our context are overwhelmingly much smaller than that limit. However, there are both soft and hard limits to the number of concurrent WebRTC connections that can exist at any given point in time, and these do carry design implications. These limits are worsened due to browsers' inefficient "garbage collection" of dead connections, causing high churn to fill the capacity with zombie connections. We explore these limitations in further detail within this section.

WebRTC uses Real-time Transport Protocol (RTP), which is in turn over UDP, with some features for media streaming and options for sequencing to detect dropped packets, bringing it a little closer to TCP than pure UDP. While the underlying RTP layer is connectionless, WebRTC still requires establishing a "connection" between two peers. To bootstrap the connection (NAT traversal / UDP hole punching) the peers must perform a handshake and send metadata back and forth through a signalling server. The time it takes to do this is not insignificant. It is an industry trope that establishing a connection between two WebRTC peers is notoriously messy and takes a while for developers to wrap their head around. Figure 3.31 shows Mozilla Developer Network's attempt at summarising the initial exchange for setting up a WebRTC connection.

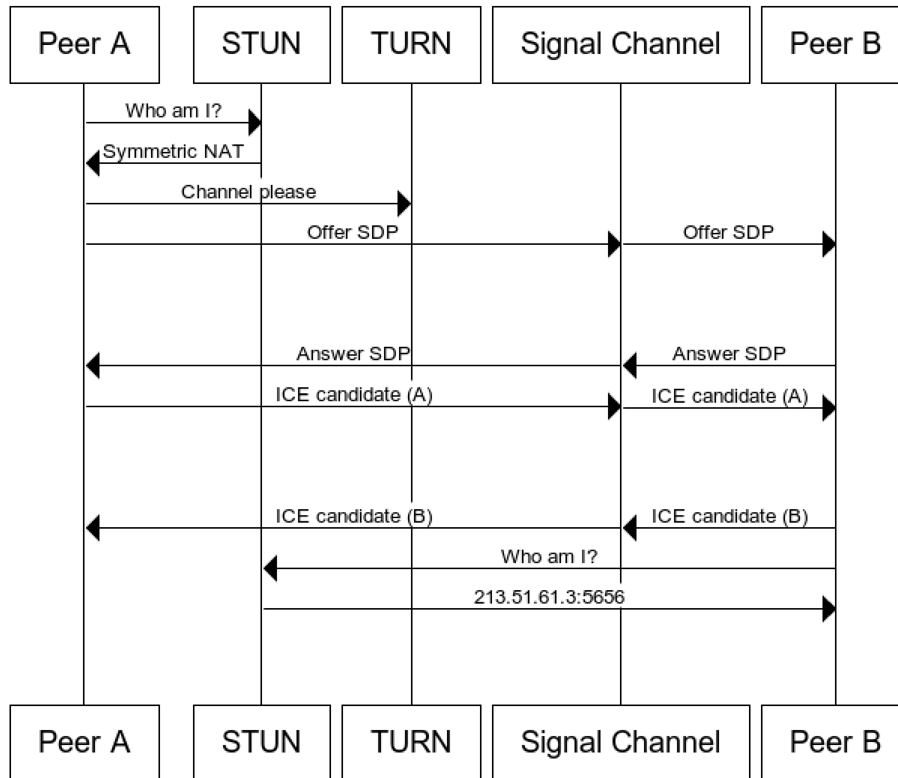


Figure 3.31: The WebRTC exchange "in a complicated diagram" by MDN[6]

Going through this handshake is far from instant, let alone predictable. For the most common WebRTC use case, video conferencing, this is acceptable because the connections last long relative to the time it takes to establish them. For a use case where connections are short-lived however, using WebRTC is a significant challenge. If not handled properly, connections are to be severed before they have even been established.

Before we can design a system around this limitation, we must first understand to what extent this poses an obstacle. We ask:

The setup for the measurements needed to answer this question is simple — we launch two peers (each running in headless Chromium instances) on the same host that runs the global signalling server at libfabric.io (more on this later). We then throttled the loopback interface by different amounts directly using the Linux `tc` (traffic control) tool. Chromiums's built-in developer tools do allow changing network conditions to presets

[6]`https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity`

such as “Slow 3G”, but these only affect the initial page load (with a bug report that has been open for many years) and are based on arbitrary definitions of what these presets mean. Further, they only limit bandwidth and Round-Trip Time (RTT).

In order to limit the effects of extraneous variables, we throttle only latency (without jitter or any temporal probabilistic modelling), and packet loss separately, in order to examine the relationship between these and the time it takes to set up a connection. This is useful because the handshake we described above for setting up is convoluted and requires a lot of back and forth and processing. Our measurements are also run completely offline — we do not use any public STUN servers. Indeed no Network Address Translation (NAT) traversal is necessary at all in this setup (and neither is the use of TURN servers) as we are focusing specifically on the overhead of the signalling/control traffic. We acknowledge however that this process may take longer if peers were to use public STUN servers, or if there are further issues in establishing the connection.

For each degree of throttling to cover different connection qualities, we repeatedly initiate and sever WebRTC connections, with one second delays between rounds. The limiting factor is the handshake/signalling back and forth between one peer, the server, and the second peer. We time this process and plotted the resulting distributions in figure 3.32.

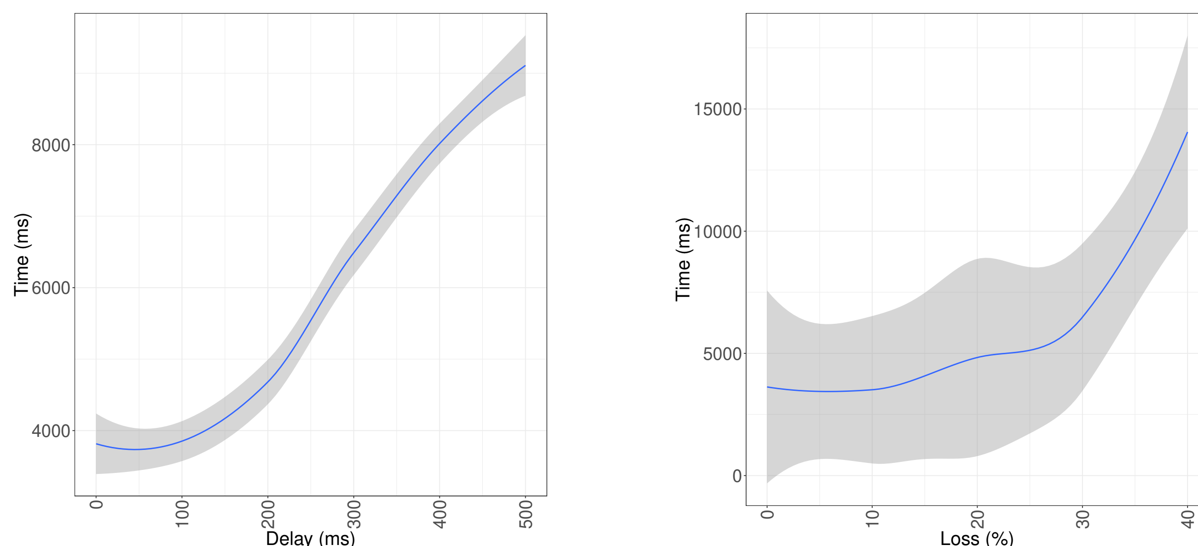


Figure 3.32: Connection establishment time overhead against latency (left) and loss ratio (right)

Here, the relationship between connection quality and the time overhead of establishing a WebRTC connection between two peers becomes evident. We can see that it is significant and cannot be ignored at either the design or the evaluation stage of this project. Later (§5), we also use these statistics to make establishing connections between

artificial peers more realistic, while keeping the simulations lean and minimising the effect of other variables we are not interested in.

Design requirement (17)
Topology must account for WebRTC connection establishment time overhead patterns

Evaluation requirement (9)
Evaluations should measure updates lost due to WebRTC overheads

There is another, more nuanced limitation to P2P connections in the browser. There are both soft and hard upper limits to the number of connections a peer can have and reliably maintain. To design a system that keeps these below those thresholds, we must understand:

What are the limits to the number of connections across browsers/devices?

The question of connection limits has been shrouded in mystery for the past decade, with conflicting ambiguous information online. As there are only very few instances of developers using WebRTC for (small scale) P2P games, and all other use cases are generally video conferencing, it is a question that is rarely asked. We clear this up by investigating the limits both experimentally and by checking Chromium and Firefox source code.

Our experiments on freshly installed browsers across generally cap out at 500 simultaneous connections per page. This is consistent across desktop and mobile devices. Unlike Chrome, Firefox's maximum is less predictable but generally in a similar range.

As of 2020, Chrome does have an actual, undocumented, hardcoded limit of 500[7]. Before this was implemented, there was no limit, going as far back as the WebKit days[8].

The reason this limit is hard to test for however is due to browser issues. Both Chrome and Firefox have always been bad at the "garbage collection" of dead connections, causing high churn to fill the capacity with zombie connections. Checking `chrome://webrtc-internals` (or the Firefox equivalent: `about:webrtc`) will often show

[7] https://chromium.googlesource.com/chromium/src/third_party/+/master/blink/renderer/modules/peerconnection/rtc_peer_connection.cc#144

[8] https://chromium.googlesource.com/chromium/src/third_party/+/5a7f431061f415177cf391511d1331c5bd9b7773%5E%21/WebKit/Source/modules/peerconnection/RTCPeerConnection.cpp

a build-up of these that count towards the 500 limit. As WebRTC was originally intended for video conferencing (where there are relatively few, persistent peers) this issue is much more significant for our use case as opposed to the primary use case.

These connections persist until they are manually destroyed, or the associated pages are closed/refreshed. One way to work around this is through one's own heartbeat implementation or using the signalling server to notify of peers disconnecting such that other peers can destroy their connection. The downside to this is that it requires a persistent connection to a signalling server. Since we require this anyway, we can get better management of dead connections for free.

Design requirement (18)
Topology must limit the number of per-peer connections

3.11 Summary

We collected a set of area traces from an NVE fitting our use case for analysis in order to answer contextual questions. After explaining how we collected these traces and how we analyse them, we proceeded to identify areas of certain "genres" — static and dynamic. We showed that these are representative, and that area distribution in our selected NVE match the findings of previous research.

We then focused on player activity over time, where we showed scale and diurnal patterns, as well as how the relationship between the number of players in an area and the mean AOI density changes based on the type of area. For static areas, this relationship was much steeper. We also examined occupancy across areas to show the existence and extent of hotspots, illustrating that avatar distribution is rarely uniform.

We then tackled questions on P2P topology churn. We first looked at churn due to motion, in the form of idle behaviour across area traces. We qualified this and showed how it differs across static and dynamic areas, then went deeper to identify any different player motion classes. We concluded that we can treat all players as belonging to a single class, and derived a finite state machine that models player motion for each area type.

We expanded this to encapsulate an analysis of player motion by location, showing that this too is not uniform and heavily dependent on the map. We also introduced a method for crowd detection in this context to understand crowd behaviour across static and dynamic areas, concluding that the effect of exploiting this for churn reduction for P2P networks is limited.

Since cheating is an important consideration when designing P2P systems for NVEs, we next examined the prevalence of this. We first presented vulnerabilities we found in our selected NVE, the proceeded to scan our dataset for instances of guaranteed cheating. We went further by demonstrating the ability to cheat at different layers within the context of this game by implementing a browser extension that enables the user to easily perform these.

On that note, we examine the rank data from the profiles we scraped, and to what extent this can be used as a proxy for reputation (for the purpose of eventually using reputation as a metric in P2P neighbour selection). We compare this against more well established social network metrics that we extract from in-game gift-giving behaviour. This tells us that rank can be used as a reputation proxy, while eigenvector centrality can form a "ground truth" for evaluative purposes. We find that other metrics, such as account age, are weakly correlated to reputation.

We then establish that heterogeneity is significant from developer analytics, and proceed to examine the constraints that browsers pose to P2P networks. Here we measure that the overhead of establishing WebRTC connections is relatively severe and will have a big influence on the design of our P2P system such that it is resilient to this overhead as caused by topology churn.

Throughout the measurements made in this chapter, we drew out a set of design and evaluation requirements from the results of these analyses which answer the questions we posed all in pursuit of research question ④ (see §1.3). The following is a collated list of these requirements.

Design requirements

① Topology must perform well under static conditions

② Topology must perform well under dynamic conditions

③ Any ALM must accommodate a range of AOI densities

④ Topologies cannot assume constant activity/density and must accommodate cyclic changes over time

⑤ Topology must scale to a minimum typical volume of players (replication, connection limits, etc)

⑥ Topology must account for concurrent sparse and dense regions

⑦ Topology must consider high peer ephemerality

⑧ Topologies must account for churn caused by dynamic active times

⑨ Topologies can assume a single player motion class

⑩ Topologies cannot rely on crowd behaviour for churn reduction

⑪ The integrity of messages in transit must be guaranteed

⑫ Network must be resilient to deliberate dropping/delaying of forwarded packets

⑬ Misrepresentation of neighbourhood trust/reputation metrics must be effectively mitigated

⑭ Neighbourhood selection metrics should incorporate rank as a reputation proxy

⑮ Topology must account for heterogeneous devices with varying network limitations

⑯ Topology must account for heterogeneous pairwise links at the network level due to geography

⑰ Topology must account for WebRTC connection establishment time overhead patterns

⑱ Topology must limit the number of per-peer connections

Evaluation requirements

① Area size and number of players in a synthetic trace must be set in such a way that realistic AOI densities are emulated

② Synthetic workloads must emulate realistic occupancy distributions

③ Synthetic traces must reflect peer ephemerality

④ Synthetic traces must emulate realistic active/idle states

⑤ Synthetic traces can assume a single player motion class

⑥ Performance must be measured under workloads with some uniform motion flow

⑦ Eigenvector centrality can be used to assign bad actors realistically to evaluate a network's resilience to cheaters

⑧ Evaluation must be over heterogeneous networks

⑨ Evaluations should measure updates lost due to WebRTC overheads

In the following chapters, we seek to meet these requirements in design, implementation, and evaluation, with the confidence that these are founded on strong empirical justification.

Chapter 4

P2P Update Dissemination

4.1 Overview

In the previous chapter, we have presented an extensive range of measurements that capture Networked Virtual Environment (NVE), browser, and network requirements. In a nutshell, our system must be *adaptive*, *scalable*, *secure*, and *browser-fit*.

In this chapter, we describe the design and implementation of our system, as well as the libraries and APIs we released, driven by the requirements and constraints of the previous chapter. We begin by outlining and justifying our high-level approach, then describe the system architecture and algorithms in detail.

4.2 Decentralisation: to what extent?

Peer-to-peer (P2P) topologies either need to form emergently, or have to be computed somewhere. It is tempting to try and distribute topology computation too, as a completely distributed system has a certain purity to it that can be attractive to a systems engineer. We must however be careful not to decentralise for the sake of decentralisation, but to instead motivate this for every component of the system. As we have discussed in §2.3, applying the main tenet of Software-Defined Networking (SDN) to a P2P context, and separating the control and data planes, is uncharted territory with a lot of potential.

We have alluded to the use of a "signalling server" in the previous chapter. In its simplest form, this is just a communication channel that is used to bootstrap WebRTC connections, and it cannot be avoided. It usually also acts as a lobby or directory of peers that can be connected to — akin to DNS for P2P.

Right off the bat, we have a system component that cannot be distributed. Indeed, the same holds true for any P2P system. Take for example BitTorrent files. While the actual file transfer is certainly decentralised, actually finding peers relies on trackers servers, which the torrent file points to; the DNS of file sharing. Incidentally, these tracker servers are usually how peers downloading copyrighted content off of public trackers get in trouble.

Through content-based addressing (in the case of BitTorrent: magnet links and Distributed Hash Tables (DHTs)) it is possible to decentralise peer discovery, and we discuss this in more detail in the coming sections. However, it is not possible to decentralise the discovery of these URIs (in the case of BitTorrent: torrent search engines). Blockchain use cases (e.g. the Ethereum Name Service[1]) or distributed storage (e.g. IPFS' InterPlanetary Name System[2]) are all use cases that have different incentives besides availability, such as preserving privacy and/or mitigating censorship. These are incentives that do not necessarily apply to NVEs and therefore do not warrant the same level of decentralisation.

This begs the question: is it truly necessary to decentralise every part of an NVE? As discussed in §2, we already know that 80% of traffic is non-critical state updates (e.g. chat and position updates) [87]. If our goal is to distribute update dissemination such that this traffic does not flow through a centralised server, to raise performance and reduce cost, then it would be foolish to push for decentralising any more than that, especially without a clear motivation and introducing a litany of design and architectural challenges.

We emphasise that the goal here is not decentralisation for the sake of decentralisation. We already know that we need at least one server for control and orchestration, so we can add to that critical update forwarding (low-traffic) and other control layer activity.

There is one such control layer activity that many existing solutions try to distribute: neighbour discovery. There is no reason that our signalling server cannot also compute the topologies and tell the peers how to connect. Meanwhile, there are many reasons not to distribute this function; challenges in coordination and mitigating against bad actors to name a few. The information used to make these computations is, of course, stale due to the latency between peers and signalling server, however it would be variably stale if distributed, and the signalling server at least has a full overview of all peers, so it can send more comprehensive control messages.

We later verify in our implementation that signalling server control traffic is indeed negligible compared to update traffic between peers, as expected. Incidentally, centralising topology computation also has positive commercial and intellectual property impli-

[1] `https://ens.domains/`

[2] `https://docs.ipfs.io/concepts/ipns/`

cations, although these are tangential to the academic implications. We do not consider this as an element of our system design per se (which we present in the next chapter), as the focus is more on update dissemination, but it is still important to consider the effects of centralising topology computation. This raises an important question however: how frequently should P2P topologies be recomputed?

We can attempt to answer this question by first finding out how frequently the metrics that neighbour selection depends on change, primarily motion. Figure 4.1 shows the distributions of movement intervals across areas. This data was collected by simply logging the inter-arrival times of position updates from any peer.

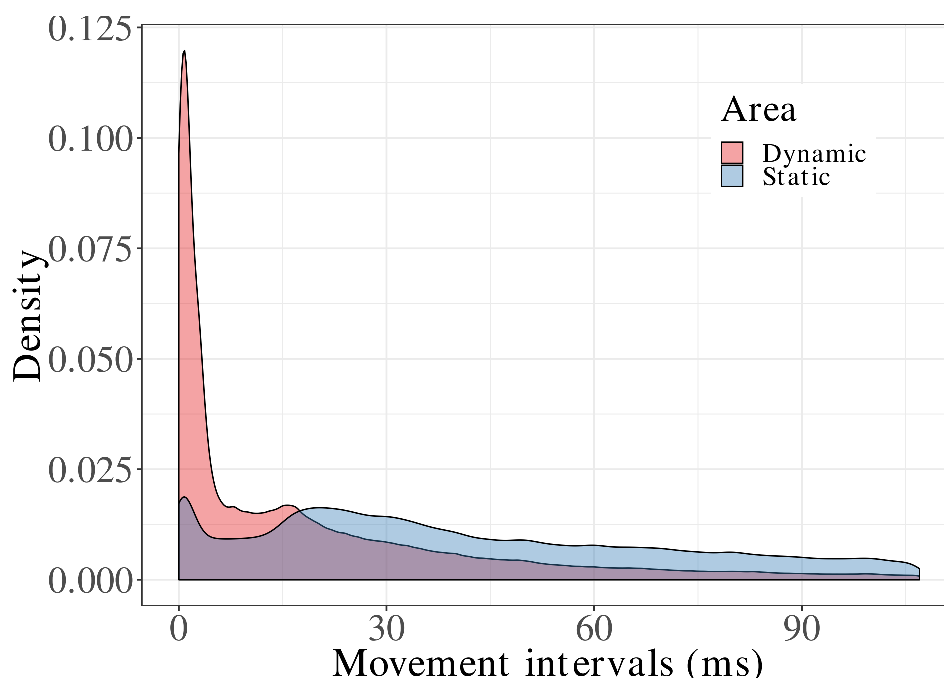


Figure 4.1: Density plots of movement intervals (between all peers) across areas, capped at the 90$^{\text{th}}$ percentile

At first glance, this does not bode well. Especially for the dynamic trace, movement is frequent enough that we are on the order of a few milliseconds. Imagine a theoretical "worst" topology ever, where a single tiny motion causes all existing connections to sever and an entirely disjoint set of connections to be established. Figure 4.1 would imply that if the peers do not figure out the new topology in a matter of milliseconds, the topology is almost guaranteed to be very sub-optimal consistently.

If we consider that topology updates are actually very localised and that peers do not care about those outside of the topology outside of their own Area-of-Interest (AOI), then this requirement becomes more relaxed. In §3.4 we have established that update intervals at the dynamic area are around 700 milliseconds. Also, in §3.5 we have established that

AOIs can be up to around 20 players dense. This means that in the worst case, where these updates are perfectly spaced out out-of-phase, we can still have movement intervals of up to 35 milliseconds.

Fortunately, movement does not necessarily mean that a topology needs to update. This is naturally dependent on the topology — a completely connected network only needs to update when players spawn/despawn for example. In the system we present in the next chapter, topology update interval is treated as simply a parameter that can be configured.

In §5, we investigate this in further detail however in order to determine what a good value for this parameter is. It is conceivable that this parameter can be dynamic; a function of churn measured in real time and adapt to reduce the number of connections/disconnections at every recomputation. This is surely a future research goal, but currently too tangential to the focus of this thesis on Update Dissemination (UD).

4.3 Architecture

From the reasoning of the previous section, we decide to centralise topology computation. We are of course limited to WebRTC as this is currently the only way to communicate P2P in browsers. Fundamentally, we have two logical components: the signalling server and the peer. Figure 4.2 shows these components at a high level and lists the roles of the signalling server.

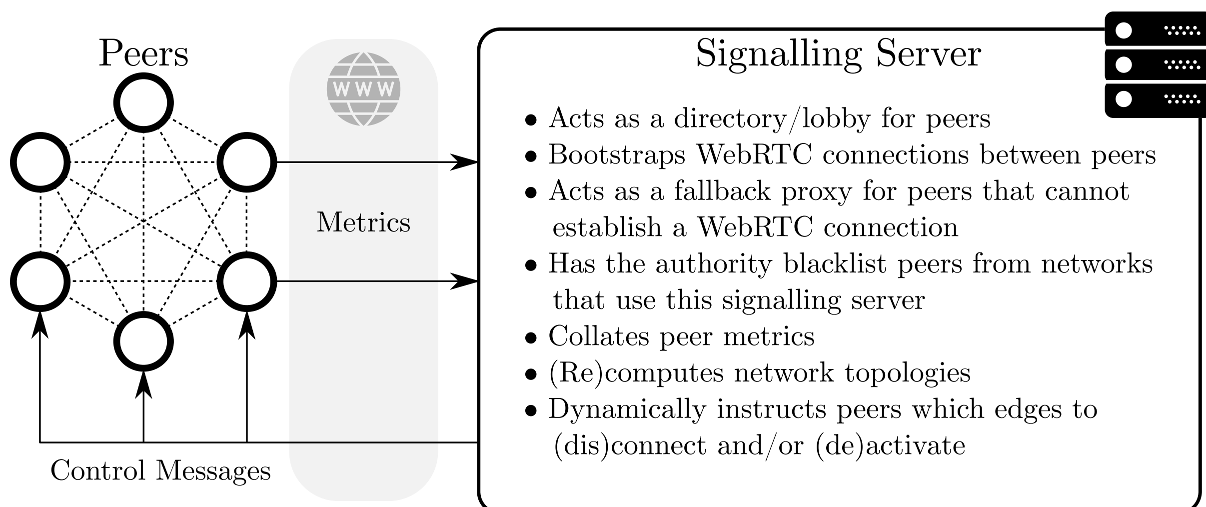


Figure 4.2: Our high-level architecture listing the roles of the signalling server

The default roles here are what we have described in the previous section; enabling peers to connect. We get to control *how* these peers connect "for free" by having the signalling server double as a topology orchestrator. Based on information that the server gets from peers over time, it can compute topologies and dynamically instruct individual peers which other peers they should connect to or disconnect from, as well as which connections should be (de)activated. We will go into more detail on this in the next section.

For some topologies, collecting metrics is entirely optional. Some topologies can also be very static, so once a peer is connected, they do not receive any subsequent control messages. For certain use cases, all peers join at once (for example, a game match between a fixed number of players) and in cases like this, once the signalling server has connected these peers, the control channel between peer and signalling server can be severed entirely. Usually it is still beneficial to keep the connection between peer and signalling server alive however, as the signalling server can much more accurately inform peers of a peer disconnecting (see the end of §3.10 for more information on this phenomenon).

We would like to stress that the browser library[3] (which we discuss in more detail later), that we have built as a result of this work, is completely topology-agnostic. We implement a range of common topologies that can be configured based on developer needs, and only include our own as one of these options. Indeed, by default, our method only kicks in when more than eight peers are connected to the same namespace — below that, the topology is completely connected. We also allow developers to drop in their own custom topology computation functions. The following sections describe our own design, then we revisit the browser library and signalling server API in §4.8.

4.4 Routing within our system

It is important that we discuss the life cycle how messages are handled on top of the overlay network topologies that we build. In this section, we will make some references to topologies that we have implemented and will describe in §4.8.2. This is due to the fact that UD can vary somewhat from topology to topology, but for the most part, it is built on the same flooding-based approach across topologies.

In §2.7.3, we discussed different routing strategies that can apply to Application-Layer Multicast (ALM). For structured networks, such as DHTs, these are generally straightforward — once DHT routing tables have been constructed, the obvious routing

[3] `http://libfabric.com`

scheme is to send packets to the peer with the hash closest to the target. For unstructured networks, there are a range of geographical routing protocols relying on a myriad of heuristics that can be used for single-path routing.

In a UD context, standard flooding is the default strategy. It would make no sense to sequentially unicast updates to individual AOI peers. Indeed updates cannot be encrypted due to this because these systems rely on receiving updates and forwarding them such that all peers that an update crosses paths with get updated. We note that more advanced flooding strategies exist [125] but these are beyond the scope of our work. We describe how messages are routed over the topologies and systems we implement when it is relevant, although often topologies only allow specific paths, making the notion of routing redundant.

To better explain how updates are disseminated in our networks, we will first introduce some terms. When a node either receives or emits an update, that update follows a certain life cycle. We summarise this life cycle in figure 4.3.

An update can either be generated by a peer, or have been received by that peer from another one. If it was received, the peer first checks if it is outdated, i.e. if the peer has already received newer information in the past. If it has, that update is ignored. This also prevents broadcast storms caused by cycles in the network, as peers will not forward an update they have already seen a second time.

New updates are accepted and the local state of that update's originating peer is modified. Topologies that do not require update forwarding (e.g. Complete or AOI) stop at this point (the dotted arrow in figure 4.3). These one-hop topologies can often send updates over suboptimal routes as path A–B–C may be faster than path A–C. All other topologies check to see how many hops the update has already made. If this exceeds a threshold, the update is once again dropped. This is common for flooding-based propagation to prevent endless propagation.

We set the threshold at 3 hops as a precautionary measure, but also log when an update is dropped as a result of this (update age). Most topologies generally have an upper-bound that is either hard (e.g. Superpeer topologies) or soft (e.g. Ring topologies). Theoretically, this threshold can be changed in real-time as the number of peers changes, however this is currently beyond the scope of our research.

Following this, the update is cast, as are any updates that are emitted by the peer itself. Recall that we are dealing with ALM, so after this point, the update is replicated and sent to multiple recipients.

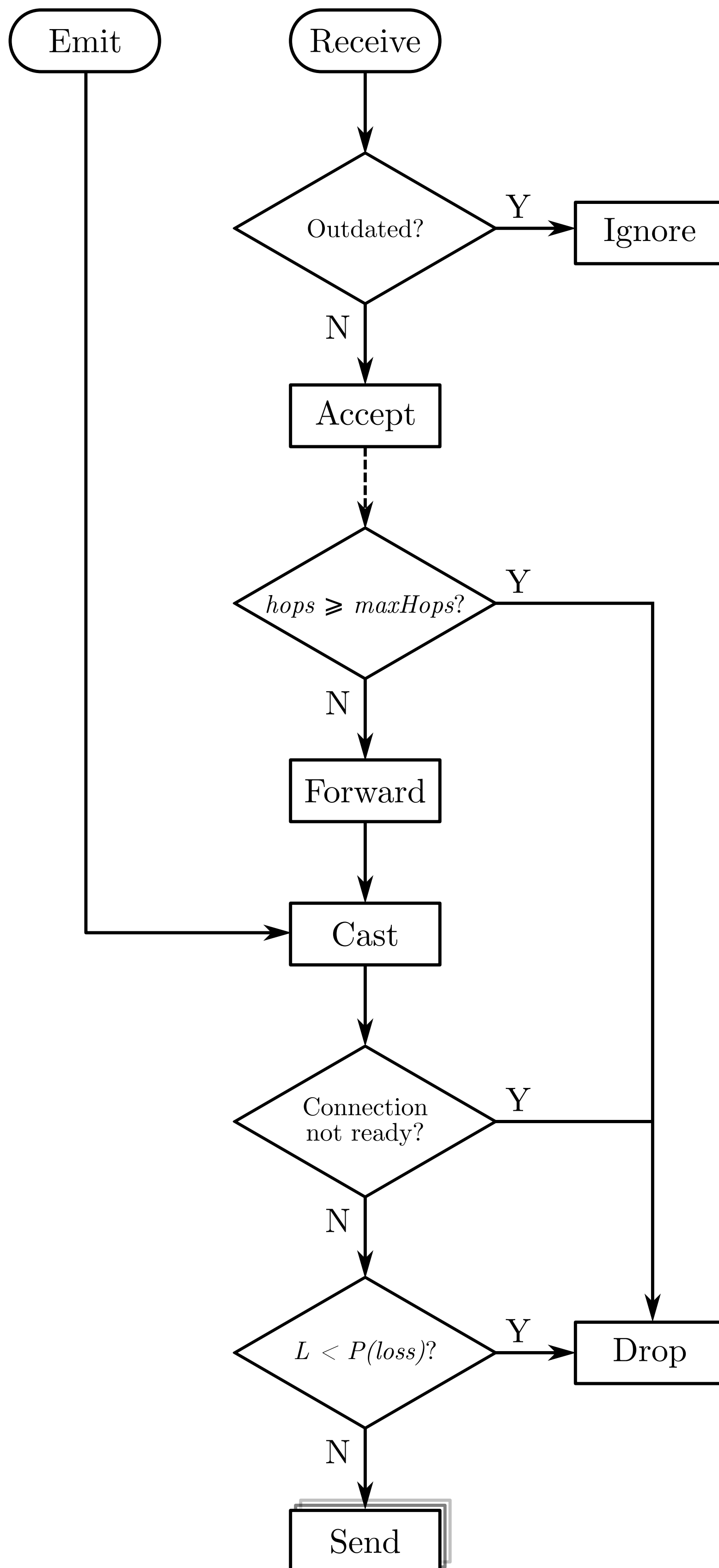


Figure 4.3: Life cycle of an update

There are some constraints on which peers can even be sent to that generally apply to all topologies. These are:

- Peers do not send updates to themselves
- Peers do not send updates back to where they came from
- Peers do not sent updates to the originator
- Peers do not send updates to unconnected peers
- Peers do not send updates down inactive links
- As long as the above conditions are met, all traffic to superpeers is allowed
- Peers do not send updates to peers outside of the originator's AOI for topologies where this is not allowed (all topologies except Ring ones and Ours)
- For topologies where the above constraint holds true, but in the rare case where a peer does not know what the originator's last known position is, the update is dropped

The last condition will only happen if a non-position update (e.g. interaction or chat) precedes the first position update and does not encode within it the originators position, therefore making it impossible for the forwarding peer to determine if the recipients are within the originator's AOI. Updates are also not sent if the WebRTC connection is still in the process of being established, which we later discover can take a significant amount of time. There is no point in buffering updates as they are so short-lived that, by the time this queue is flushed, they are already irrelevant. The final point in the life cycle at which an update may be dropped is when it has been transmitted but dropped due to packet loss. We include this here as we will refer back to this figure in §5.2.2 of our evaluation, where packet loss is artificially induced with a deterministically random probability based on a predefined loss ratio.

4.5 Our algorithm

In §2, we have established that most existing solutions generally take the same approach — completely connecting peers in a certain virtual AOI. This means that the peer network topology is dependent on how these AOIs are defined, which in turn is usually dependent on virtual peer positions.

We approach this problem from a different, general angle. In this section, we begin by defining the problem we solve more explicitly. Then we formally delineate the details of our solution while justifying our design decisions, and how our method can be extended, throughout.

In chapter 2 we also established how most existing solutions only look at Virtual Environment (VE) user positions for Interest Management (IM) and therefore how the peer network topology is computed. Our solution also considers positions, however these positions are not raw VE positions, but rather n-dimensional coordinates that are computed based on a variety of default factors, which we call distance metrics. We select four initial pairwise metrics, which we will unpack in more detail shortly. These are:

- **Virtual distance** — the virtual distance in the VE between two avatars
- **Network latency** — the network distance between two peers
- **Trust score** — a "reputation" distance between two players
- **Time connected** — the time two peers have been connected

As we focus on distributing update dissemination, we relegate peer network computation to the centralised signalling server as discussed earlier. The data transmitted to do this is comparatively low in volume, however our architecture can be fully distributed in the future, in a similar fashion to Vivaldi [28]. In current architecture, the above metrics are collected by a coordinating server which then computes a topology and instructs peers on which other peers to connect to or disconnect from.

The above metrics can be extended to include arbitrary factors, however this has some limitations. The first limitation is that these metrics have to satisfy the triangle inequality. This is often assumed, for example with Network Coordinate Systems (NCSs) such as Vivaldi [28], although practically this may not be true, simply because of sub-optimal routing on the internet and heterogeneous routes. For example, with nodes A, B, and C fully connected, the path from A to C may be faster through B than a direct path. We investigate how we can mitigate this issue through increasing the dimensionality of the final coordinates.

The second limitation is that the metrics have to be pairwise/undirected. A simple distance measure is the same both ways, however a trust score is subjective (and again may violate the triangle inequality). In order to overcome this limitation, we combine directed metrics by using their mean, max, or use a different statistical average depending

on the metric. These two limitations also enable important optimisations which we discuss later.

Figure 4.4 illustrates the steps of our method from the collation of distance metrics to instructing peers to update their connections. In this section, we discuss each step in this pipeline in detail.

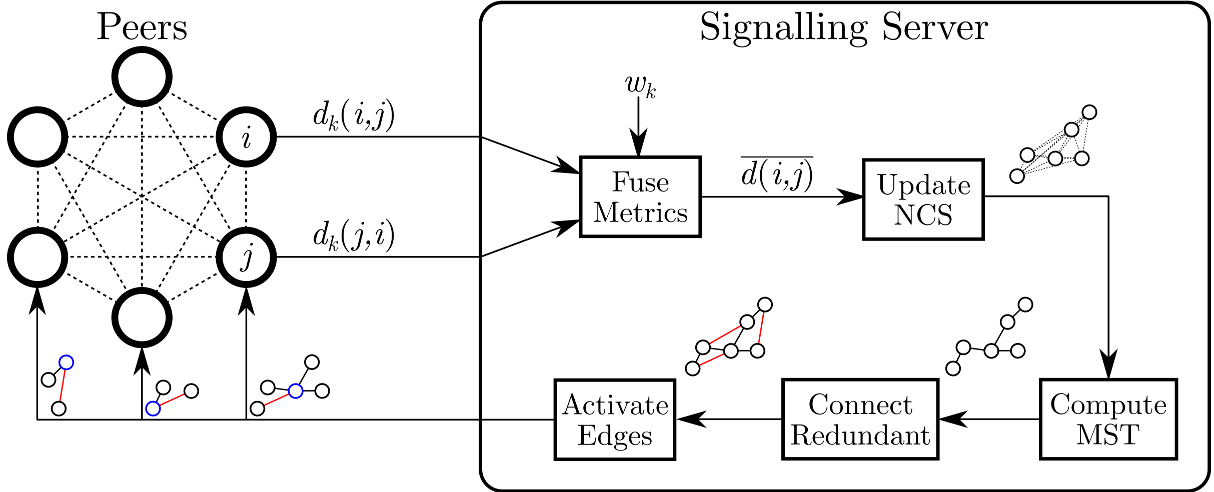


Figure 4.4: Our peer network topology computation pipeline

4.5.1 Distance metrics used

Each of the above four factors are computed on a per-edge basis. In this section we describe how we compute these.

4.5.1.1 Virtual distance

To capture the first factor, we calculate the euclidean distance between the VE positions of nodes i and j, which we call x_i and x_j respectively. These coordinates can have an arbitrary dimensionality, M (usually 2 or 3), so we denote the dimension as m, i.e. $x_i = (x_{i1}, x_{i2}, ..., x_{iM})$. We refer to this variable as $d_p(i, j)$ defined in equation 4.1.

$$d_p(i, j) = \sqrt{\sum_{m=1}^{M} (x_{im} - x_{jm})^2} \tag{4.1}$$

4.5.1.2 Network latency

To compute the latency factor, we simply divide the round-trip time between a pair of nodes, rtt_{ij} for nodes i and j, by two. This does assume that the latency one way is the same as the latency back, which is not necessarily true practically, but this is common assumption that is made in related work [28]. We refer to this variable as $d_l(i, j)$ defined in equation 4.2.

$$d_l(i, j) = rtt_{ij}/2 \tag{4.2}$$

4.5.1.3 Trust score

Each node i maintains a trust score for every other j denoted as t_{ij}, where $0 \leq t_{ij} \leq 1$. t_{ij} is better thought of as as *distrust* score, as we invert it here to save the step of inverting it later (since more trust is analogous to a smaller distance between peers). We initialise this directed trust score at 0, trusting peers by default, and it changes dynamically over time.

Trust is computed automatically based on how frequent a peer sends an incorrect state. As peers have no way of telling if another peer is behaving maliciously or simply have a corrupted game state, we compare state to that sent by other peers and mark the majority as "correct". If a peer continues to bad state, then they become less and less likely to be kept as a connection, or have new connections initiated with them. This can be overridden by a peer manually marking another peer as untrustworthy (e.g. through higher-level observations such as game-layer cheating) which can set this factor to 1 or an extremely high number, effectively associating a huge cost to connecting to this peer.

Our system is versatile enough to allow other trust/banning mechanisms to be encoded within it in a similar fashion (such as sharing trust scores with other trusted peers), however the trust metric we describe serves as a basic proof-of-concept that encoding application-critical, directed factors into the topology computation decision can be done.

Note that t_{ij} is not necessarily equal to t_{ij}, making it a directed factor. To transform it into a pairwise/undirected factor, higher distrust takes precedence. This makes sense intuitively, as it allows peers to override malicious peers who try to artificially paint themselves as more trustworthy. We call this transformed variable $d_t(i, j)$ defined in equation 4.3.

$$d_t(i, j) = \max\{t_{ij}, t_{ji}\} \tag{4.3}$$

4.5.1.4 Time connected

Finally, a common problem that has plagued previous work is the frequent switching of connections. Practically, there is a cost associated with connecting to a new peer, as the initial handshake adds overhead latency. Some approaches have tried to minimise this overhead by attempting to create topologies that are much more stable over time as peer positions change [14]. There has even been work that characterises the way that players move in a VE in order to anticipate how this motion will affect churn, and this information can be used to augment how the network topology is computed [67, 91].

While both of these areas are important, the first can be applied to any system that uses Delaunay triangulation, so it is not an alternative to our system but a potential future avenue of optimisation, and the second is very application-specific. We instead use concepts from the load balancing space to solve this problem in a simple, general way. It is not uncommon for load balancing systems to adopt a notion of "stickiness" when routing, in order to add some temporal consistency to links and prevent sporadic switching between two similar queues or sinks.

We therefore capture the time a connection between two peers has been active, s_{ij}, as an additional factor when computing our network topology. This is a straightforward pairwise measure (identical to s_{ji}), however its effects should not be linear, as it makes more sense to have a kind of exponential backoff where new connections are more likely to stay and older ones are more equivalent.

As such, we use a simple sigmoid function, the hyperbolic tangent, over the time connected, so that it can be treated as a linear factor in the next step of the process. This can be adjusted to make the connections more or less "sticky", so we include the constant c_s to denote this. This constant controls how fast a connection converges in stickiness to other older connections, *not* how much the stickiness variable as a whole affects the topology computation. A lower constant means that this will happen faster (less sticky).

We also reiterate that these factors can be the result of arbitrary computation that should affect the topology in some way. We call this "stickiness" variable $d_s(i,j)$ defined in equation 4.4.

$$d_s(i,j) = \tanh\left(\frac{s_{ij}}{c_s}\right) \tag{4.4}$$

Finally, we define the full set of pairwise distance metrics as $d(i,j)$, which for our implementation is made up of $\{d_p(i,j), d_l(i,j), d_t(i,j), d_s(i,j)\}$, but can be extended ar-

bitrarily. For all these metrics, where $i == j$, the value is 0, and the pairwise/undirected limitation will mean that swapping i and j will yield the same value (e.g. $d_t(i,j) == d_t(j,i)$).

4.5.2 Building an appropriate coordinate system

As our distance samples may be sparse, we simulate a force-directed graph to fill in the missing edges. Given a graph with weighted edges, we can simulate a spring system that computationally/iteratively moves nodes to optimal positions with minimised energy. The nodes position themselves in such a way that missing samples can be estimated through euclidean distance. This is equivalent to Multidimensional Scaling[27].

Crucially, this is done for each distance metric, *not* for the final combined metric. This is because different metrics may require different simulation parameters based on their volatility.

In the case of network distances, our model is very similar to Vivaldi [28], with some minor differences for nodes to be positioned more uniformly. In addition to this, we do not limit the computed coordinates to just three dimensions — while this may be sufficient for NCSs that are close to physical coordinate systems, other coordinate spaces may naturally require a higher dimensionality (especially to mitigate triangle inequality violations).

This can be expressed as an optimisation problem, where we seek to minimise the energy of the system based on a cost function that takes the different factors into account, however describing it as such can be quite indecipherable. In this section, we therefore aim to explain it in a more intuitive manner.

Whenever a new node is sampled, it is initialised with a random m-dimensional coordinate. Recall that we have defined our distance metrics as $d_k(i,j)$ for metric k and nodes i and j. For every sampled edge, we use this value as the resting length of a spring between these two nodes. Let $d_e(i,j)$ denote the euclidean distance between two nodes. δ is the magnitude of the displacement of the spring from its relaxed/resting state. We calculate the restoring force vectors F_i and F_j that this spring exerts on each node at any given point in time (equation 4.5) using Hooke's Law.

$$\begin{aligned} \delta &= d_e(i,j) - d_k(i,j) \\ v &= x_j - x_i \\ F_i = -F_j &= -\sigma\delta\hat{v} \end{aligned} \tag{4.5}$$

Here the constant σ denotes the spring coefficient, which is related to spring "stiffness" and controls how fast a spring will reach its final state. Normally this is called k, but we already use that as a distance metric placeholder. The unit vector of v simply gives us the direction of the displacement, and the result is inverted because the restoring force acts in the opposite direction of the displacement. Each node can perform this calculation independently for each of its neighbours.

We also use Coulomb's Law to give each node, which we treat as particles, a small "electrical" charge. This is important specifically due to the potential sparsity. This is analogous to the universal law of gravitation, so another way to imagine it is each node having a negative mass and a negative gravitation, like repelling magnets. Crucially, the force decreases quadratically with distance, as the inverse square law applies in all cases. Equation 4.6 describes the resulting forces.

$$\begin{aligned} v &= x_j - x_i \\ F_i = -F_j &= \frac{Q_i Q_j \hat{v}}{d_e(i,j)^2} \end{aligned} \tag{4.6}$$

Here Qi and Qj are the charges on each node. For our purposes, all charges are the same, so the expression $Q_i Q_j$ can be replaced with a global charge becoming Q^2. We also disregard Coulomb's constant (an additional coefficient) from the equation completely, as "charge" is just a metaphor here, and the particles are not actually moving through a medium.

We also make sure that the distance between two nodes is never zero (otherwise theoretically the resulting force is infinite) by adding a small distance if that is ever the case. Similarly, we apply a maximum to the force that can be applied at any timestep. We assume a constant timestep, however this would work with a dynamic timestep too as long as the forces are multiplied by the elapsed time since the last simulation step.

An immediate observation is that adding these charges breaks the distributability of this algorithm. This is true if the simulation calculates the effect of the charge of one node on all other nodes. We therefore limit the effect of charges by AOI radius. As a result, any given node only needs to know about its neighbours and the nodes in its AOI for this algorithm to be distributed.

This is a fair simplification to make as (in part due to the inverse square law) distant nodes exert very little force on a given node, just like when simulating the motion of planetary bodies, astronomers will not take into account the negligible gravity of far away stars. It also would not scale at all if every node had to perform these calculations

with global knowledge of every other node for virtually identical results.

Finally, we add the spring forces to the charge forces to compute the final force acting on a node, and we treat that as an acceleration. We do not include a notion of "mass" (or rather, all nodes have a mass of 1), so the extent of this acceleration is controlled indirectly by the constants σ and Q. We apply these accelerations to the nodes every timestep, and over time, the graph will converge to a state with minimized energy (where the nodes move very little), after which we stop the simulation.

In other words, the code for this physics simulation involves simply repeatedly iterating through all nodes, calculating the sum of forces acting on this node (via its edges and AOI nodes), and updating its position in this semantic space based on the acceleration resulting from this force. In the distributed variant of this algorithm, each peer would perform its own calculations and update its own position. The simulation stops when it converges on a stable state (i.e. the nodes stop moving or move exceedingly little).

We repeat the simulation every time a new sample is made, which corresponds to a new spring. Since the "disturbance" to the spring system is only ever one node at a time, it converges in very few iterations. In Vivaldi, smaller values of t (a constant that controls the step size for moving nodes) are preferred. Other than for network distances, we generally require larger values for the equivalent — our graph coordinates need to be more dynamic because our collection of metrics has the propensity to change much more rapidly, meaning the graph coordinates need to reflect that much more aggressively. Meanwhile Vivaldi, which only focuses on network distances, tends to deal with much more static spaces, as network distances on their own change slower or are time-averaged.

We select constants that are as high as possible, such that the simulation can converge quickly, without causing the system to become unstable. Here "unstable" means a system that is too responsive and springy for a timestep that is too large relatively, where positions can end up oscillating indefinitely and never converging for example.

The last step is to normalise and combine the metrics we have described into a single metric that we can use as edge weights for constructing our P2P topology. To do this, we apply a weighted average. Each weight effectively dictates how much influence single metric has on the positioning of a node within our final combined coordinate system. These weights are referred to as w_k in equation 4.7 and map directly to $d_k(i,j)$ for K weights/metrics. These weights are normalised. The final combined metric, the edge cost, we call $\overline{d(i,j)}$.

$$
\begin{gathered}
\overline{d(i,j)} = \sum_{k=1}^{K} w_k d_k(i,j) \\
\text{where} \sum_{k=1}^{K} w_k = 1
\end{gathered}
\tag{4.7}
$$

It is also important that this coordinate system is euclidean. Previous work has shown the advantages of using non-euclidean models, however this would make our system very difficult to scale, as we rely on this property when computing spanning trees in the next step. Euclidean Minimum Spanning Trees (EMSTs) allow optimisations in calculation that brings our method to a level of efficiency that is practical.

4.5.3 Computing topologies

The final step in our process is computing a topology that fits our constraints over the nodes with the computed coordinates. This is an NP-hard problem, so our solution is a heuristic which we later compare to optimal solutions found through brute force.

Seeing as sparsity is a strong requirement for P2P networks in the browser, we start by computing the provably most efficient configuration: the Minimum Spanning Tree (MST). Here, our edge costs correspond to $\overline{d(i,j)}$. Over time, we compute the edge costs for every pair of distinct nodes and build a complete graph of edge costs. When we physically connect two nodes in the network, we say that the edge has been *activated*.

An MST creates many bridges however, which makes our network very fragile; a single node disconnecting or a single link failing can disconnect the network graph. To remedy this we add connections using algorithm 1. Here, `peers` is a list of peers, each containing a list of edges, each associated with a cost $\overline{d(i,j)}$.

The result is an MST network with added connections to bring the k-edge-connectivity of the network above a certain predefined threshold. This is not optimal, but we compare the resulting network to bruteforced optimal topologies.

We modify half the cost of edges that connect two under-provisioned peers, as activating these kills two birds with one stone by adding an edge to two nodes at once. We also recognise that a minimum k-vertex-connectivity may be a constraint that more closely maps to resilience, and we intend to include this in our algorithm in the future.

```
Data: peers
Input: minK, the minimum edge-connectedness
Result: A network that is efficient and resilient
First, compute the MST using Prim's algorithm and activate MST edges;
foreach peer of peers do
    /* Ignore adequately connected nodes                                  */
    if peer.activeEdgeCount < this.minK then
        /* Half cost of edges connecting 2 weakly-connected peers        */
        foreach edge of peer.edges do
            if edge.peerA.activeEdgeCount < minK and
             edge.peerB.activeEdgeCount < minK then
                edge.modifiedCost = 0.5 * edge.cost;
            else
                edge.modifiedCost = edge.cost;
            end
        end
        Sort peer.edges by modified cost;
        /* Activate cheapest edges to adequately connect this peer       */
        foreach edge of peer.edges do
            if not edge.isActive then
                activate edge;
                if peer.activeEdgeCount ≥ minK then
                    break;
                end
            end
        end
    end
end
```

Algorithm 1: Our heuristic algorithm for adding resilience to peer networks

It is also trivial to modify this algorithm to favour nodes that are less connected. Similarly, we can add an upper limit to the number of edges a node can have. The reason for both of these is to ensure that our topology is well within browsers' WebRTC connection limits. We later find however that this is practically unnecessary and redundant (§5).

4.6 Pre-connection outside of AOI

In §3.10 we emphasised the WebRTC connection overhead and how this makes dynamic P2P incredibly difficult in the browser. In addition to our longevity distance metric, we have found a way to minimise the effect that this has.

We know four things at any given point in time: (i) the maximum movement speed of a player, (ii) the AOI radius, (iii) pairwise latencies, and (iv) the expected WebRTC connection establishment time overhead distribution (via the measurements from §3.10). Our solution to do away with these overheads entirely is to pre-connecting with peers within a margin outside of an AOI to "pre-heat" connections, and hot-swapping them in and out. This margin is based on those four factors.

The calculation is straightforward. The connection "cooldown" for a pair of peers at any given point in time is a function of their pairwise latency. We take this cooldown time and simply multiply it by the maximum speed of a peer. This tells us how much distance a peer can cover in the worst case scenario while a connection is being established. We take this value and double it, as motion is relative and the actual worst case is peers running at maximum speed *towards* each other. Then we simply extend the player's AOI by a margin equal to that distance, creating an extended AOI in which we pre-connect any peers that happen to cross it.

The obvious disadvantage to doing this is that we end up with many more connections than if we limit them to only within the AOI. Some of these connections are never even used, as the player might decide to move in the opposite direction after all. It is possible to mitigate this by taking into account which direction a player is facing, or predicting their motion based on past motion, but we have found this to be unnecessary. This is due to the fact that our topology is already as sparsely connected as possible and that we perform better than other topologies in this regard, let alone stay under browser thresholds. In §5 we show just how effective this technique is.

4.7 Mitigating cheating

Our P2P approach introduces other non-negligible challenges. In a client-server context, there is a stark trade-off between how much game logic the server carries or attests (which increases server costs) and players' ability to cheat by modifying the client. An example of this are games where all physics simulation is done client-side to minimise lag and a player can modify a client to manipulate their position and clip through solids.

In a peer-to-peer context, this problem persists between peers, however the cost of validating game states falls to the clients. As no authoritative servers exist, peers have to decide either manually or autonomously to disconnect and/or blacklist cheating peers by maintaining an array of IPs of past offenders seen first-hand. We have captured these requirements in §3.7.

On the spectrum between MST and completely-connected, we cannot therefore simply go for MST as it is more efficient. We must introduce redundant connections for two reasons: (*i*) resilience and (*ii*) accountability. Figure 4.7 shows a visual example of this where peer ④ relies on just peer ③ for updates, as peer ③ has a good connection to the left side of the network. We add redundancy by connecting ④ to ① also.

For the first, it is imaginable that a node goes offline for whatever reason, and a new MST must be computed. To avoid the overhead and potential lag in repairing the peer network, redundant connections are an advantage. For the second, if a peer relies on only one other peer to update their global game state, that peer can spoof the game state. While every peer can perform their own validation for impossible game states, or states that imply cheating, there are edge cases where this becomes less obvious when a peer receives realistic but divergent states from two or more other peers.

It is therefore important that we work in a mechanism for preserving the integrity of messages as they are forwarded between peers. An obvious approach would be to encrypt messages end-to-end between peers. There are not necessarily just two "ends" here however, as the peers that forward these updates need to inspect the information to update their own states. States are multicast and propagate through the network rather than being iteratively unicast.

We therefore instead add a 2-byte signature fingerprint to every message. These messages are signed using Ed25519[4] which is well known to be incredibly fast (hundreds of messages per second on modern hardware) and secure. Public keys are provided by the signalling server while bootstrapping WebRTC connections together with other peer metadata.

Is a 2-byte signature fingerprint in any way resistant to preimage or collision attacks? Definitely not, but we have the advantage that these messages are incredibly short-lived, on the scale of milliseconds, and become obsolete when the next message is sent. The high frequency of these messages makes any kind of bruteforcing pointless and the class of attacks by forwarders that modify the messages are effectively mitigated, meeting our requirements on cheating.

4.8 Implementation

We implemented the above system in two parts. The first is the signalling server, and the second is the client/peer library which runs in the browser. Additional documentation

[4] `https://ed25519.cr.yp.to/`

for our browser library and signalling server API is available at `http://libfabric.com`. We have already used this library to create three real-time P2P applications:

- **Camwire** (`https://camwi.re`) — "The Pastebin of video conferencing"; focus on ease of use (no installation or login) and any room can simply be created/joined by navigating to `https://camwi.re/some-room-name`
- **Drawire** (`https://drawi.re`) — Collaborative whiteboard, similar to above
- **Filewire** (`https://filewi.re`) — P2P file sharing

These applications are tangential to this thesis, and only exist to demonstrate the practicality of our implementation. NVEs are however a fundamentally different use case and have vastly different requirements to other use cases. For example, video conferencing has historically been small-scale (one did not expect to have hundreds of participants in a video call) and have much lower churn. This landscape has however been rapidly changing as of the COVID-19 pandemic. In the past, supernode architectures have been successfully used in this context, while these do not make sense in the NVE context as we explore in the next chapter. Even in the large-scale live-streaming/broadcasting context, the literature is comprehensive and P2P architectures optimise for different factors.

Our library has not yet seen widespread practical use, primarily because we have only very recently released the first public version. In our system evaluation, we focus on this implementation however, in order to demonstrate that it meets the requirements we set out to satisfy. In this section we describe this implementation in detail, from both the client-side as well as the server-side.

4.8.1 Signalling server

For uniformity, our signalling server is, like the client library, also in Javascript (Nodejs) with a handful of pure-nodejs dependencies. Our implementation is open source under an MIT license and available on GitHub[5]. A publicly accessible deployment of this lives at `https://sig.amar.io` for testing and `https://libfabric.io` for production. At the time of writing this thesis, there is no authorisation or API key system on top of the signalling server endpoints, so we simply prefix namespaces (which we describe shortly) with the domain/origin to separate applications. Future plans include augmenting the signalling server API to require developer API keys. This is of low academic importance, but necessary if this work were to ever be monetised in the future.

[5] `https://github.com/yousefamar/p2p-sig-serv`

We use the built-in Node.js `http` module along with `socket.io`[6] to create a WebSocket server on a configurable port (8090 by default, but can be changed through the `PORT` environment variable). Peers connect to this server and the server sends them a unique ID on successful connection. Peers should wait to receive this ID before any subsequent communication with the signalling server, or else they will be ignored.

4.8.1.1 Room namespaces

Peers can join namespaces, which we refer to as “rooms”. These are slightly different from what one would understand as a room as they can be nested. The room “tree” is delimited by slashes, not unlike a directory tree. For example, peers in the `/io.amar/players/moderators` room are also in the `/io.amar/players` room.

At the top level, rooms separate applications from one another, but even within the same application, developers can use these namespaces to further segment their users in some way. An example of this would be would be to create distinct areas/zones (we touched on the prevalence of this in §2.4.1) or for example add an extra AOI layer on top of default topologies. For example, it is not uncommon for games to use grid-based AOIs where peers in one cell are interested in peers in surrounding cells, and they can join and leave these cell rooms as they enter and exit. Peers can also listen for events higher up in the namespace path (for example the room for their wider 3×3 grid cell area) or indeed global, game-wide events.

4.8.1.2 WebSocket events

The following is a description of the WebSocket events that the signalling server listens for.

sdp / ice — Session Description Protocol (SDP) and Interactive Connectivity Establishment (ICE) messages that are forwarded to a user ID defined in the event body. If the destination ID is not defined or invalid, the message is ignored. These are specific to the WebRTC handshake (see figure 3.31 in §3.10). The details of WebRTC are out of scope here, but we include these for completeness' sake.

join — a message to join a room of a room ID defined in the event body. These rooms can be arbitrary strings. If a room does not exits, it is created. If the room ID is not

[6] `https://socket.io/`

defined, the message is ignored. When joining a room, a peer also joins all rooms within which this room is nested. In the completely connected topology mode (which we will discuss shortly) all peers in the same room are notified when another peer joins the room (the event is simply forwarded/broadcast to those peers). Otherwise, a topology recompute is triggered. The event body also includes peer metadata such as ID, initial state (position), and public key (for verifying the integrity of updates).

hail — a message from one peer to another peer to notify it of its existence. The destination user's ID is defined in the event body. If the destination ID is not defined or invalid, the message is ignored. This message is sent on receiving a `join` event and prompts the joining peer to initiating a WebRTC handshake (see above) with the hailing peer.

roomcast — as the name suggests, when this event is received, it is simply forwarded/broadcast to a room ID defined in the event body. If the room ID is not defined, the message is ignored. The source user's ID is appended to the event body before it is forwarded (this is true for all events however). This event is only used when peers are in client-server mode, which we will discuss shortly.

leave — a message to leave a room of a room ID defined in the event body. These rooms can be arbitrary strings. If the room ID is not defined, the message is ignored. All peers in the same room that are connected to the leaving peer are notified of this peer's departure (the event is simply forwarded/broadcast to those peers). Leave events are also fired whenever a client simply disconnects (WebSocket) from the signalling server (these can be ignored client-side however).

There is one more place, outside of the above event listeners, where the signalling server can send WebSocket messages to clients, namely whenever a topology recompute is triggered. This happens on `join`/`leave` events, but also when the topology is recomputed periodically. When this happens, the signalling server sends **topology** events to the clients. Crucially, the event body does not contain the entire updated topology, but only the deltas that are relevant to a receiving client.

The `topology` event body contains up to four arrays. The first is the `connect` array which instructs a client which peers to initiate connections to by ID. These are peers the client is not already connected to. The peers it is meant to connect to receive the same information, and whomever `hail`s second is ignored. The second is the `disconnect` array which are the delta disconnections. The third and fourth are the `activate` and

`deactivate` arrays, which instruct the clients of delta activations/deactivations of connections that are already connected. If a client somehow receives an `activate` message regarding a peer to which a connection does not exist, that peer is `hail`ed, and the connection is activated once it is established. If a client somehow receives a `deactivate` message regarding a peer to which a connection does not exist, that message is ignored.

4.8.2 Supported topologies

The method we described in this chapter computes topologies that are well suited to large-scale NVEs such as Massively Multiplayer Online Games (MMOGs). We recognise however, that different applications, and even genres of games, may have different requirements. We therefore implemented a wide range of existing and common P2P topologies that can each be used based on a developer's requirements. We also implemented APIs to allow developers to build in their own algorithms for computing topologies, as well as a framework for evaluating them. This incidentally also allows us to easily compare these different methods, which we do in section 5.

Small-scale topologies are for environments with discrete rooms that can support a number of players usually in the single digits. Examples are collaborative editing or simple First-Person Shooters (FPSs). Large-scale topologies can support a theoretically infinite number of players and are well suited to continuous MMOGs and location-based Augmented Reality (AR) games. Medium-scale topologies sit somewhere in the middle as they mitigate a lot of the scalability issues of small-scale topologies. These can potentially be suited for Real-Time Strategys (RTSs), Multiplayer Online Battle Arenas (MOBAs), or Battle Royale games.

In §2.7.1, we organised the different topologies from literature into a taxonomy with four overarching categories. We implement these high-level topologies, in addition to ones that significantly diverge from the overarching topology type. We also implement several versions of our own solution to compare against these. In this section we summaries our implementation of these such that there is a point of reference when we refer to them later.

To better facilitate the process of designing and debugging new systems and topologies, the evaluation framework we developed has some additional visualisation options for outputting video mosaics of the topologies being tested by generating graph dumps of these networks at fixed intervals. As long as these JSON dumps are formatted properly, they can be fed into a supplementary set of scripts we developed that will turn them

into snapshots (using a library called `cytosnap`[7]), convert those frames into videos (using the popular command-line tool `ffmpeg`), and finally tile those videos into mosaics (also using `ffmpeg`). Figure 4.5 is a frame from one such video over a synthetic workload. Node positions correspond to player in-game virtual positions, but this too can be easily configured to instead follow network coordinates or other layouts (e.g. a ring layout).

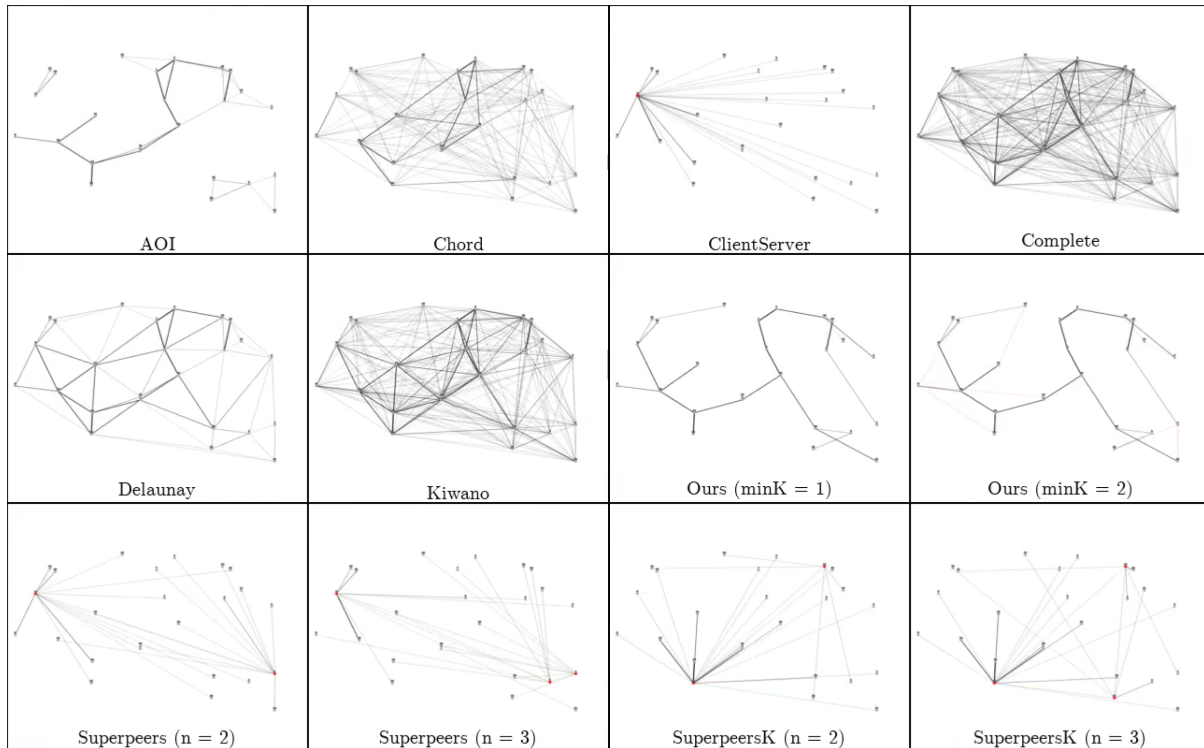


Figure 4.5: A frame from the visualisation of different topologies under a synthetic workload. Link thicknesses correspond to link quality, red links are redundant (ours), and red peers are designated superpeers for superpeer topologies.

4.8.2.1 ClientServer

Just to serve as a point of comparison, one of the topologies we implement is a simple client-server setup with the signalling server acting as the game server. All client nodes connect to this server node and the server forwards traffic from clients to other clients in its area of interest. This is less of a topology, and more of a debug flag that is disabled in production. Here, the `roomcast` WebSocket event that we described earlier is used.

[7] `https://github.com/cytoscape/cytosnap`

4.8.2.2 Complete

We also implement the naive, completely connected mesh approach to P2P networking. This is very straightforward; all peers get connected to all other peers. Forwarding is disabled for this topology (AOIcasting is one-hop).

4.8.2.3 AOI

The most common P2P UD topologies by far are AOI-based ones. Connecting to peers in a player's AOI is the most "obvious" solution, however these methods differ in how they define AOIs. Some are linked to the use case of a specific game/application and will e.g. define grid AOIs or dynamic AOIs when closer avatars are more important. Others, such as pSense [118], connect additional "sensor" nodes on the periphery to better detect when peers join AOIs, for the purpose of distributing discovery too (as opposed to just UD).

As we have reiterated throughout this thesis, we believe AOIs should not be prescribed by the system, but rather the system should accommodate game design choices. We therefore implement a standard AOI topology that takes as a parameter an AOI radius defined by the game it is used in. Forwarding is disabled for this topology (AOIcasting is one-hop).

4.8.2.4 Ring

Structured topologies (e.g. DHTs) are of course popular approaches to scalable P2P. These are significantly less popular in the NVE space which prefers unstructured overlays, but are still important to consider. In our original taxonomy, we included a range of general-purpose structured approaches for completeness, but here we focus primarily on ring topologies, as these are the only ones that have actually been used in an NVE context in any major publications. Technically, it was in the context of Object Management (OM) rather than UD, however by implementing this topology we can investigate if it would be suitable for UD while we are at it (spoiler: it is not) and draw conclusions about the use of DHT-based topologies for UD in general.

One important caveat is that these topologies are assumed connectionless and the networks are directed. In practice, each node maintains routing tables and forwards packets to other nodes efficiently (usually for the purpose of searching/storage). It really does not suit the context here, but we implemented the browser UD equivalent of this.

Like Chord, we map peers in a ring configuration based on their IDs and connect them

to peers that are powers of two away from them. WebRTC is not exactly connectionless however, and these connections are two-way, meaning the resulting topology is undirected and closer to Kademlia than Chord through this symmetry. Nonetheless this serves as a good model for this class of topology. Nodes under this topology are able to forward outside outside of their AOIs.

4.8.2.5 Delaunay

Delaunay triangulation-based topologies can have some variation. These are often nested into other techniques, for example connecting nodes by zone into disconnected Delaunay graphs. Fundamentally, these are similar enough to plain Delaunay triangulation (especially in terms of scalability) that our implementation of connecting all peers in a Delaunay graph approximates these well and later provides useful insights on how these perform.

4.8.2.6 Kiwano

The UD topology of Kiwano [32] is the third power graph of a standard Delaunay network, so it can be considered a Delaunay triangulation-based topology. Despite this, we decided to implement it because it is different enough that its performance significantly differs from simple Delaunay triangulation. It is also one of the most recent contributions to this field and the "state-of-the-art" in many ways, even though the paper does focus a lot on a different layer of scaling.

4.8.2.7 Superpeers

As already explored, many common P2P architectures for games ca be generalised to superpeer architectures. Usually this is in the form of a single player acting as "host" for a game and becoming the de factor server, forwarding the traffic for other peers. Our implementation of this is quite simple. The n peers that have been connected the longest act as superpeers and the remaining peers are divided among these in a deterministically round-robin fashion. The superpeers are completely connected among themselves. n can be controlled, and at the lower bound, this approximates a client-server architecture, while at the upper bound, this approximates a completely connected network.

Superpeer architectures do have some special rules however. For example, players can send/forward packets to superpeers even if they are outside of their AOI, since there is the possibility that that superpeer is the only path to forwarding to a different peer that is

inside their AOI. Without taking any kind of reputation into account, superpeers can also deliberately delay or drop packets in order to gain an unfair advantage in-game (unlike an authoritative server). When superpeers go rogue, the network can suffer dramatically, which is something we explore later.

4.8.2.8 SuperpeersK

Since superpeer topologies (especially the ones within our taxonomy) often base superpeer selection on factors that are outside the scope of our work (such as peer hardware resources), we generalise this into a second type of superpeer topology. This employs a resource-aware superpeer selection method: we perform k-means clustering over the peer NCS with n centroids, and select the nodes nearest to the final centroid positions to become superpeers for the peers in their cluster. This is to represent superpeer topologies with more optimal layouts and — besides our own topology — is the only class of topologies that considers network conditions explicitly in the neighbour selection decision.

4.8.2.9 Ours

Finally, the implementation of our topology within our evaluation framework is the very same we described in §4.5. Our method takes $minK$ as a parameter to control the level of connectivity and therefore fault-tolerance of our topology.

Of course our method has features that have nothing to do with the topology (for example, update integrity guarantees and pre-connecting peers). These are all turned on automatically when this topology is being used.

4.8.3 Client library

The peer library is, naturally, in Javascript, with no external dependencies. It is open source under an MIT license and available on GitHub[8]. It can also be installed through NPM as *p2p-peer* and used as a Node.js module with common browser bundlers such as Browserify and Webpack, or as an ES6 module. Additional documentation is available at `http://libfabric.com`.

For the sake of maximising the ease of development and minimising the developer barrier to entry, we encapsulate all networking functionality behind (*i*) event emitters and (*ii*) shared objects that sync between peers automatically. These shared objects are

[8]`https://github.com/yousefamar/p2p-peer`

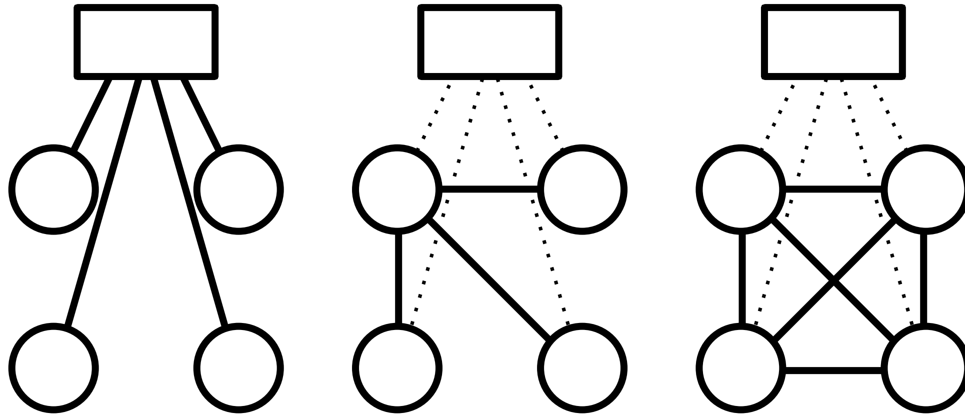


Figure 4.6: Three networking topologies of interest between servers (rectangles) and peers/clients (circles) – client-server model (left), hosted P2P (middle), and full P2P (right)

optimised for common game networking needs, for example by allowing developers to mark coordinates for interpolation and/or prediction. This abstraction is also important because we want to avoid code duplication and maintaining strongly related code in more than one place. This is common in client-server applications where changes to client communication requires changes to server communication in parallel, and vice versa. With our system, developers need only maintain peer code.

We also recognise that some applications will simply never require a topology as advanced as ours. In some cases, especially with few nodes, it is enough to simply completely connect all nodes, or have a single superpeer. We therefore allow developers to select these simpler topologies instead. Developers can of course augment our system with their own topologies too or use any of the topologies we have implemented and described in th previous subsection.

For example, in the middle option in figure 4.6, a peer is designated "host" and acts as a de facto authoritative server[9], as opposed to a completely connected topology such as the right one in figure 4.6. This limits the server costs of the game provider to simply acting as a lobby/directory for finding these rooms/groups, but at the same time, the number of players that a host can support is more limited than a standalone server.

Each still require a server for signalling purposes to exchange the information required to establish a connection between peers. This low-traffic server connection only needs to be maintained if the peers need to be notified when a new peer joins their network, which translates to joining their game room/area/instance/arena/match/etc. Otherwise it can be severed after a peer network is set up. To use this hosted P2P model, a developer

[9]Incidentally, there are several valid reasons for developers to use this architecture, and these have been explored in the past by popular games [86]

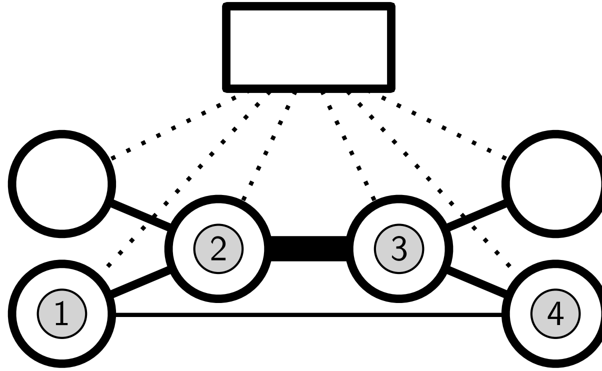


Figure 4.7: An example of a computed MST topology where peers with better connections (② and ③) act as supernodes, and with redundancy (① and ④)

need only configure their clients to use our Superpeer implementation with an n value of 1, then disconnect peers from the signalling server entirely once all players are connected for a match for example.

Our library exposes a `PeerNetwork` object which handles peer connections and emits events based on different events that happen in the network (such as peers connecting or disconnecting). We allow arbitrary namespacing through *rooms* of which a peer can join multiple. We will avoid detailing the various APIs we make available as these are well-documented on the GitHub repository, but will provide a high-level overview here.

Our library exposes a set of methods that can be called directly if required. These are:

- `signal(event, ...args)` – Sends an event and data to the signalling server
- `join(roomID)` – Joins a room with a particular ID
- `leave(roomID)` – Leaves a room with a particular ID
- `broadcast(event, ...args)` – Sends an event and data to all connected peers
- `async connect(sigServURL)` – Connects to a signalling server

`PeerNetwork` also has three properties; `ownUID` which is the peer's own UID in the network, an array called `peers` which contains instances of `Peer` for all connected peers, and an array called `rooms` which will be explained shortly.

One can `.disconnect()` from or `.send(event, data)` directly to each `Peer`, and on the other side, peers will emit events based on what is sent that can be listened to. These events do not need to be defined anywhere beforehand.

As an added layer of abstraction over network events, each `room` contains an `eventEmitter` `syncedData` object which implements an Observer design pattern such that any changes made to this object (or any nested objects at any depth) emit an event distinct to a property's path in this object. The value, along with the path of the added/modified property's path (e.g. `.foo.bar`) is propagated through the peer network and all peers in a room can expect to see the property at that path updated. For added efficiency, we only broadcast data that has been modified (aka delta-coding) which is a common multiplayer game optimisation. We also keep track of which data has been set by which peer for data ownership and to prevent broadcast storms in the peer network.

Finally, we add configurable, game-specific optimisations over this in the form of state interpolation/prediction. This is quite simple in our implementation. For position interpolation, we override the position getters/setters to instead set a target position, then simply linearly interpolate the actual position to that target over time. When the actual position gets within a configurable distance threshold to the target position, it snaps to that position. For prediction, we extrapolate from the current position, the previous position, their timestamps, and the current game time, to naively predict where avatars would be if they continued moving based on these samples. A more advanced motion prediction approach would be to take more samples into account and use better prediction algorithms (e.g. a moving average on velocity, a Kalman filter, or even machine learning-based approaches utilising mobility traces). We relegate this to future work as we have the mobility traces to investigate this largely uncharted research area.

Namespacing within the object and how rooms are organised is left to the developer to define. As mentioned before, rooms can map directly to an arena instance in a MOBA context for example. Another possibility is rooms mapping to a 2D or 3D spatial grid where players join and leave rooms seamlessly as they navigate through the game environment, prioritising connections to the players who are closest to them in the game.

Similarly, rooms can be "nested" into a tree structure for optimisation purposes, such as for limiting update frequency for peers that are further away from each other in-game (e.g. position updates) creating Network Levels of Detail (NLODs). This is especially powerful in an MMOG context.

Chapter 5

Evaluation

So far, we have captured Networked Virtual Environment (NVE), browser, and network requirements through an analysis of collected traces and targeted experiments, and presented the design and implementation of our system that aims to capture these requirements. This chapter describes the evaluation of our system in depth, with respect to different workloads, parameters, and existing solutions, in order to demonstrate that we have indeed captured these requirements and met specific research goals.

5.1 Overview

We begin this chapter by outlining the evaluation questions we seek to answer. To frame the chapter, we summarise the results of these questions through table 5.1. Then we discuss our evaluation setup, what our performance metrics are, and justify the parameters and workloads we use. Each section after those aims to answer targeted evaluation questions. At a higher level, these evaluation questions seek to demonstrate that we have met the research objectives set out in §1.3 under the requirements captured in §3. We cross-reference these requirements where necessary and reiterate them. These are denoted by circled numbers corresponding to each evaluation requirement (e.g. ①).

5.1.1 Evaluation questions

The following is a list of four evaluation questions we seek to answer in this chapter, each corresponding to a later section. Answering these questions under different workloads and environments is implicit, although the focus is, unsurprisingly, on scalability. We elaborate on what our performance metrics are in subsequent sections.

1. **Scalability** — How does our system scale with number of peers, compared to others?
    - How does it scale in terms of consistency metrics?
    - How does it scale with respect to browser connection constraints?
    - How does it scale with respect to browser/device bandwidth constraints?
2. **Churn sensitivity** — How sensitive is our system to peer/connection churn, compared to others?
3. **Loss resilience** — How resilient is our system's performance to packet loss, compared to others?
4. **Cheating mitigation** — How well does our system mitigate cheating, compared to others?
    - How influential are cheaters in our system at different proportions?
    - How does cheating affect the performance of our system at different proportions?

We summarise the results of these questions through table 5.1. Here, we have highlighted the best results in green, acceptable results (e.g. still below browser constraint limits) in yellow, and the worst results in red. This table serves both to summarise our evaluation results throughout this chapter, as well as highlight how our system is clearly the most versatile and well performing under these criteria. Since we have linked these criteria to the use case of Peer-to-peer (P2P) networks for browser-based NVE Update Dissemination (UD), it follows that our system is best suited to this context as it strikes a balance across workloads and performance metrics.

Table 5.1: Summary of evaluation results with best results highlighted green, acceptable yellow, and worst red

Topology	Performance Scalability	Concurrent Connections	Bandwidth Requirements	Churn Sensitivity	Loss Resilience	Cheater Influence	Cheating Performance
AOI	Low drift Med miss Low–med extra Med drift at scale High miss at scale Med extra at scale	Low scalability	High Low at scale	High High at extremes	Med	Low	Somewhat affected
Ring	Med drift Med miss Med extra High drift at scale High miss at scale High extra at scale	High scalability	Very high High at scale	Low–med Low–med at extremes	Low	Very high High variance	Somewhat to minimally affected
ClientServer	High drift High miss High extra Low drift at scale High miss at scale Med extra at scale	Very low scalability	Low, except server Low at scale	Low Low at extremes	Low	Unaffected	Somewhat affected
Complete	Med drift Low miss Low–med extra High drift at scale Low miss at scale Med extra at scale	Very low scalability	High High at scale	Med Low at extremes	Low	Unaffected	Minimally affected
Delaunay	Med drift Med miss Low extra Med drift at scale Med miss at scale Med extra at scale	High scalability	High Low at scale	Med–high High at extremes	High	Med	Somewhat affected
Kiwano	Med drift Low miss Low–med extra High drift at scale Low miss at scale Low extra at scale	Plateaus at approx 50	Very high High at scale	Low Med at extremes	Med	High High variance	Minimally affected
Ours (minK = 1)	Low–med drift Low miss Low extra Med drift at scale Low miss at scale Low extra at scale	Very high scalability	Low, tight variance Low at scale	Med Med at extremes	High	Low	Somewhat affected
Ours (minK = 2)	Med drift Low miss Low–med extra Med drift at scale Low miss at scale Low extra at scale	Very high scalability	Med, tight variance Low at scale	Med Med at extremes	High	Low	Somewhat affected
Ours (minK = 10)	Med drift at scale Low miss at scale Med extra at scale	—	High at scale	Med at extremes	Med	Med	Minimally affected
Superpeers (n = 2)	High drift High miss High extra Low drift at scale High miss at scale High extra at scale	Low scalability	Low, but far outliers Low at scale	Med Low–med at extremes	Low	Low until majority	Very highly affected
Superpeers (n = 3)	High drift High miss High extra Low drift at scale High miss at scale High extra at scale	Low scalability	Low, but far outliers Low at scale	Med Low–med at extremes	Low	Low until majority	Very highly affected
SuperpeersK (n = 2)	Low drift High miss High extra Low drift at scale High miss at scale High extra at scale	Low scalability	Med, but far outliers Low at scale	High Med at extremes	High	Low	Highly affected
SuperpeersK (n = 3)	Low drift High miss High extra Low drift at scale High miss at scale High extra at scale	Low scalability	Med, but far outliers Low at scale	High Med at extremes	High	Low	Highly affected

5.2 Setup and methodology

In order to evaluate our system, we need a suitable testbed. The majority of our evaluation questions are best-suited to a simulated environment, as this allows for more control when stress testing with extreme parameters, and removes extraneous influence unrelated to an aspect of the system that we are testing, yet may still affect the results.

We are however careful not to simplify our setup to the point where it can no longer be accurately compared to a real deployment. To do this, we consider as many factors as possible that may affect live performance. We then compare these results with those from a real, yet smaller, deployment. The code for our evaluation testbed and related scripts are freely accessible on GitHub[1].

5.2.1 Our testbed

Our evaluation testbed runs on several third-party compute nodes, however each experiment runs with set parameters on a single node, such that we can run several experiments in parallel. Any randomness uses a seeded pseudorandom number generator that uses the same seeds across nodes such that the experiments are consistent. All components of our experiment testbed (refer to figure 4.4) run in parallel threads, with the signalling server running in the main thread. The main components here are the workload generator, the main thread, and the peer threads. Figure 5.1 shows an overview of this and we describe each of these in the following subsections.

Communication across these components is throttled to reflect varying network conditions, which we will discuss shortly. Any randomness anywhere in our simulations is using a seeded pseudorandom number generator, reset on every repetition, such that all results are deterministic and consistent across varying parameters, workloads, and topologies. Threads are also tightly synchronised such that their behaviour does not diverge based on small differences in timing.

As our library and signalling server are written in JavaScript, targeting browsers, we opted to write our evaluation testbed in the same language, such that we can directly use the same code for computing topologies and update dissemination between nodes. This has some drawbacks in terms of speed and ease of development for this sort of programming, yet is much closer to reality.

[1] `https://github.com/yousefamar/p2p-sim`

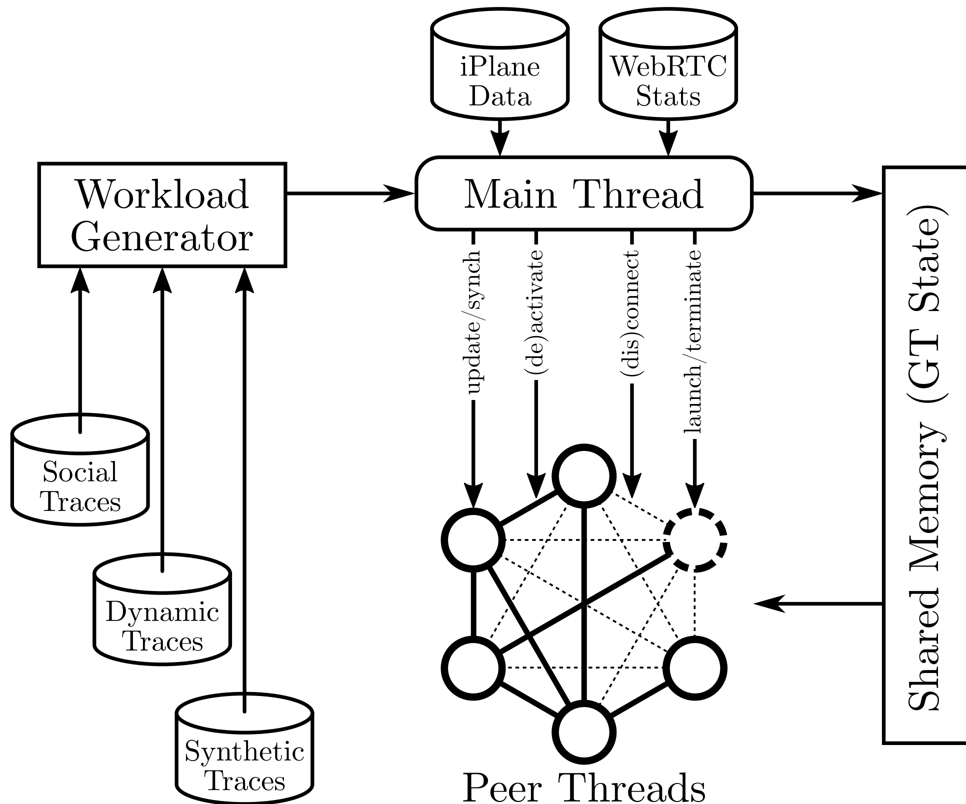


Figure 5.1: An overview of our evaluation testbed setup

5.2.1.1 Workload generator

At a conceptual level, the workload generator component of our system emits events in chronological order based on which workload is specified. We discuss what these workloads actually are shortly in §5.2.4.

At an implementation level, these are literally asynchronous generator functions[2] that either stream from disk or generate workloads on the fly. These are very easy to extend as long as they yield events in the same format defined in §5.2.4.

5.2.1.2 Main thread

The main thread launches all other threads and also runs the signalling server. The signalling server updates `SharedArrayBuffer`s with ground-truth position coordinates from the workloads. Each peer has access to these ground-truth positions in addition to their own, potentially inconsistent, view of the position coordinates of peers in their Area-of-Interest (AOI). Of course, in reality, peers would not have access to the ground-

[2] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*`

truth data without time travel or FTL networking. However this allows each node to calculate their own consistency and drift distances for later collation.

The main thread also has the task of collating all of the statistics (see §5.2.5) that each peer collected, validating and summarising these, and writing them to a series of files for later analysis. It will launch different experiments with different parameters depending on configuration controlled through command-line arguments and environment variables.

Because these experiments can take quite some time (days), even on the 16 core servers we use, the main thread writes current progress to `stdout` such that we can keep track of what experiment is currently running, with which parameters, with a progress bar to tell us how far through the workload we are. It also prints summaries statistics after every iteration so we can sanity check that the numbers are making sense while the experiment is running. Based on the configuration given, the main thread will automatically destroy all threads after an experiment, reset the PRNG, and start the next iteration under the next topology/workload/parameter set.

5.2.1.3 Peer threads

Each player "node" is run in a Node.js Worker thread via the `worker_threads` module[3]. Under the hood, each of these is a V8 isolate. Recall that this is what Cloudflare uses for their serverless platform, as discussed in §2.2. So while these are not as light as traditional threads, they have much smaller overheads than an entire Node.js instance (such as with the `cluster` module), let alone containers, which most other serverless providers use.

We have two extremes in terms of the runtime of a peer when simulating. On one end are headless browsers for each peer for realism, and on the other are all peers running in the same process. As V8 isolates are a valid abstraction for distribution, we chose to compromise and use these so that we can explore more extreme parameters (such as large numbers of peers) without the overhead making this infeasible, while still having a realistic simulation.

As each run is very CPU-intensive and has as many threads as there are peers at a time, we run our evaluations on up to four mnx.io c1.2xlarge instances at a time, which are described as having 16 CPUs and 32GB of memory. Considering that we go through the same workload many times for different topologies and parameters, this is critical in order for the simulations to run faster than real-time and minimise compute costs.

Peer threads are launched and terminated as they spawn and despawn as dictated by

[3] `https://nodejs.org/api/worker_threads.html`

the workload. The signalling server in the main thread continuously connects and disconnects peer threads with Node.js `MessagePort`s, informs them of edge (de)activations, and sends them updates that are supposed to originate from them to disseminate among other peer threads.

The individual player threads then act as they would in "production". Each thread can communicate with connected peers via Node.js `MessageChannel`s which the main thread sets up for them based on the topologies that the signalling server computes. Peers can disseminate updates through these based on logic that we describe in this chapter. Each peer also collects its own statistics, which are consolidated when a peer despawns and logged to several files at the end of each simulation round. The next section describes how we ensure that the communication along these `MessageChannel` is as realistic as possible considering that these are not actually nodes that are geographically spread out on a range of heterogeneous hosts.

5.2.2 Modelling communication between nodes

While related work often runs evaluations under ideal conditions, such as lossless networking measured only in "hops", this is not enough to evaluate how effective an update dissemination would be in the wild. Indeed we rely on this heterogeneity in inter-node networking in order to construct more optimal topologies. In this subsection, we describe how we built a model of the network underlay to be used in our experiments. To be clear, this part is entirely emulated based on real measurements — the communication channels between components are throttled to match these measurements.

In order to make our simulation of the underlying networks more realistic, we make use of historical iPlane [94] data. This is preferable to datasets from King[4] [55] and other datasets, as iPlane is the only dataset that also includes associated loss measurements.

We create statistical distributions of network measurements from the dataset over all of the years that iPlane has been collecting these (2006 – 2016), and use this to build a graph of interconnected nodes, each with an IP address. To do this, we first crawled all the online historical iPlane data (with some pointers from the original author regarding the formatting and hidden directories) and fed it into a local graph database. We also crawled the logs of inter-Point of Presence (PoP) loss ratios. While the iPlane datasets are not available in our repository, all of the code for crawling those datasets is.

For bandwidth, although it was planned, they never released inter-PoP measurements,

[4] `https://pdos.csail.mit.edu/archive/p2psim/kingdata`

only bandwidth data for iPlane nodes in BitTorrent swarms averaged by /24 prefix. This is acceptable to us however, as it is common to not make any claims about peer bandwidth capacity, and simply assume unlimited bandwidth. We then measure individual peer upload/download to draw conclusions of the bandwidth requirements under a set of constraints for a topology. If these numbers are too high to be expected to work on common consumer devices with average connections, we can make claims about the feasibility of a particular setup.

From this DB, we then selected the 10000 nodes with the most latency measurements. We further filtered these to include only ones with PoP mappings (these mappings are part of the iPlane dataset and one PoP can contain many IPs). Incidentally, these also include geographical coordinates, which we also stored, although we noticed that many of them are invalid. We note that when visualising the resulting graphs and plotting node coordinates on a geographical map (figure 5.2), clusters in the graph that have better connections with each other tend to correspond to nodes on the same continent. This is not surprising, as previous work has shown the validity of using geographical distance to estimate latencies due to strong correlation between the two (see §2).

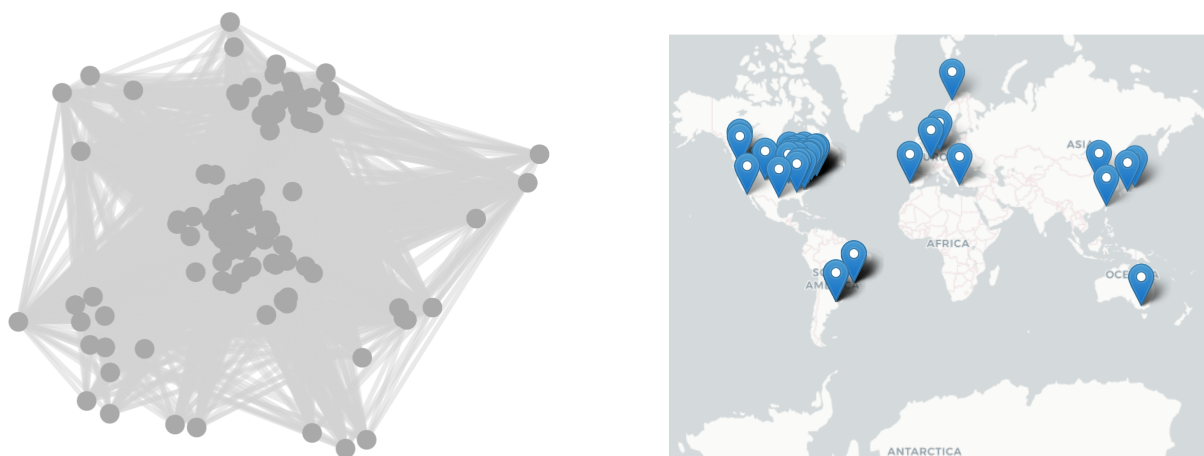


Figure 5.2: A visualisation of the iPlane network before completely connecting (left) and a corresponding map of known/valid PoP coordinates (right)

While this graph is reasonably well-connected, it is not completely connected, so we use our Graph Coordinate System (cf §4.5.2) to fill in the missing edges, such that the graph is complete. When a peer spawns, it is assigned an IP address from a pool of unused IP addresses, and its edges with other peers map directly to the network graph in terms of network characteristics. When a peer despawns, the IP address it was using is freed and returned back into rotation.

As the number of iPlane nodes with PoP mappings is limited, it is possible to run out

of IP addresses to use in an experiment. In the majority of our experiments, we have not encountered a case where the number of concurrent peers exceeds the number of available iPlane IPs. For some however, we do (and we note where that happens). It these cases, we randomly generate new nodes within the Graph Coordinate System with random IPs and estimate their latencies. This is preferable to simply reusing active IPs as this would be equivalent to two peers on the same host which is less realistic. Our simulations show no signs of degradation or deviation using these randomly generated nodes.

As previously hinted, we do not create full WebRTC connections between each simulated peer as there are too many variables to consider. Instead, we additionally include an artificial "cooldown" period to inter-peer `MessageChannel`s before these can be used for communication. The cooldown (the time it takes for a WebRTC connection to actually be established) is a function of pairwise link quality (from iPlane data) based on the measurements we took over our real implementation in §3.10. Other contexts and approaches assume a zero setup cost (rightly so, as UDP is connectionless, provided hole punching etc was successful). This is not a luxury we have.

Our incorporation of iPlane statistics is similarly straightforward. Communication between peer threads is instant, however message arrivals are queued in a buffer at the receiving node, and only processed after a delay corresponding to the network latency between that pair of nodes. Similarly, each transmitting node has a configurable probability of dropping outgoing messages entirely. If a peer despawns before it has processed the messages in its queue, the entire queue is dropped.

If a peer has no available means of propagating a message (either because it lacks the connections, the connections have not yet cooled down, or the connections are inactive), the message is dropped. An alternative policy is to queue outgoing messages until a path becomes available, however this is widely considered to provide no benefit, as these messages are very short-lived, and even small delays can make them irrelevant.

Recall the life cycle of an update in figure 4.3 and our discussion of it in §4.4. This life cycle is matched quite closely in our experiments, but some parts, such as loss and connection cooldown, are simulated and updates are dropped based on models we have built previously. Each attempt at sending to individual peers has a deterministically random chance of dropping based on the link loss ratio. Further, if the peer has been marked a cheater, it will increment a "corruption" counter encoded in the update that tracks how many times an update encounters a cheater in transit. Similarly, peers record how many corrupt packets they receive and how corrupt those were. This information is not used by the peers for decision-making, or considered in payload size, but is simply used for later analysis.

5.2.3 Topologies evaluated

The topologies we compare our system against are the very same that we implement in §4.8.2 so we skip describing these again here. We do however also compare variants of some topologies, including our own. For example, the Superpeer and SuperpeerK topologies each with values of n at 2 and 3 separately, as well as Ours with $minK$ values of 1, 2, and 10.

As mentioned in §4.8.2, our method has features that have nothing to do with the topology (for example, update integrity guarantees and pre-connecting peers). To be clear, we are not just evaluating the topologies here, but the entire system, so these features are taken into account of too.

In these experiments, the server in the ClientServer control is on the same physical host, so in order for this architecture to be as close to realistic as possible, we use an IP address from the iPlane pool to act as the server's IP address and assign it the associated network characteristics. For our experiments on mitigating cheating, the server also has some special properties, for example it is not susceptible to malicious behaviour in the same way that superpeers are for instance.

5.2.4 Workloads evaluated

Workloads in this context refer to update traces (real or synthetic) that we can play back over different P2P systems such that we can make different measurements over these systems. In this section, we describe the workloads we use throughout the experiments we run in this chapter. Recall that we captured a number of workload-related requirements over §3 which we summarised in §3.11. These were:

① Synthetic workloads must emulate realistic occupancy distributions

② Synthetic traces must reflect peer ephemerality

③ Synthetic traces must emulate realistic active/idle states

④ Synthetic traces can assume a single player motion class

⑤ Performance must be measured under workloads with some uniform motion flow

⑥ Area size and number of players in a synthetic trace must be set in such a way that realistic AOI densities are emulated

Here, all but ⑤ are requirements that explicitly apply to synthetic workloads. ⑤ can apply to synthetic workloads too, however trace workloads will meet that requirement out of the box. The remaining requirements directly inform the way in which we generate synthetic workloads for evaluation.

As introduced in §3, we have collected several month's worth of application-layer network traces, including mobility data, for two NVE areas: a slow-paced, social area and a faster-paced, dynamic area. We use these very same traces as real workloads for evaluation. In addition to this, we generated synthetic traces by using the Random Waypoint Model (RWP) mobility model which we have discussed in §2.5.4, the parameters for which we will discuss shortly.

It is crucial to use various different workloads to explore the generalisability of techniques. For example, some topologies may no longer hold up in a dynamic setting, such as an First-Person Shooter (FPS), due to the connection churn overhead. Similarly, traces may not include large spikes of virality where a system must be able to scale to an unlikely number of peers, while a synthetic trace can be tuned to extreme parameters. We therefore use this variance to cover this range of different scenarios.

To make use of the trace workloads, we first convert them into a more usable format — a CSV file with timestamped events. Other than a timestamp, each event includes at least: the source peer's ID, the number of bytes of the entire update payload, and the source peer's AOI radius. For this trace, the AOI radii are constant across peers and time, but this column is included for testing dynamic and asymmetric AOI radii in the future. The event type column is one of the following:

- **s** — A spawn event indicating that a new peer has spawned. Every time a player spawns, and connect to the signalling server, the signalling server recomputes topologies automatically (in addition to regular recomputations).
- **u** — A position update event indicating that a peer has moved. If we detect the arrival of an update before a spawn event, we automatically spawn the player at that position, as we assume the spawn event was dropped or came before the start of the trace. These lines include position coordinates.
- **a** — A update event that was not a position update (e.g. some other update type, such as an interaction or chat).
- **d** — A despawn event indicating that an existing peer has despawned. If an unseen player ID despawns, the event is ignored. When a player despawns, we recompute topologies automatically.

All updates trigger an AOIcast from the source peer. For position updates, this includes the coordinates. For any other AOIcast, we only keep track of the payload size for statistics.

Synthetic traces only have spawn, update, and despawn events. All updates are position updates; we do not consider other kinds of updates in this case, as is standard in related work.

The payload size of each event is 49 bytes, except for despawn events which are 41 bytes as they do not contain position data. Figure 5.3 shows a breakdown of the payload.

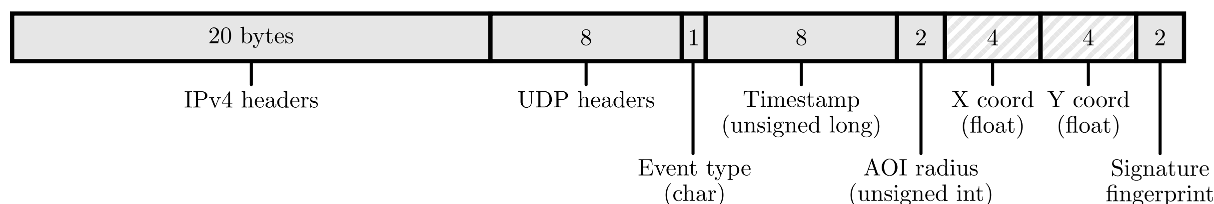


Figure 5.3: A breakdown of the synthetic trace payload

Here, the X and Y coordinates are not included in despawn events. The signature fingerprint is the last two bytes of the full fingerprint, as justified in §4.7. In cases where the AOI radius is constant and global, it is not necessary to transmit this with every package. As there is precedent for asymmetric AOIs (i.e. one player can see another but not the other way around) [107], and certainly for variable AOIs (e.g. the nearest n avatars) [9], we decided that it is important to encode this information in synthetic payloads. This payload can easily be extended to three-dimensional coordinates.

The above sizes are of course not necessarily the same across all games, however they are typical for average optimisation and it is more important that the size is consistent across all topologies tested for comparison. It is also conceivable that the number of bytes may differ based on if additional state data is sent in the same packet, if it is compressed, encrypted, or signed in different ways.

Location updates are also only sent if they change. Modern games will delta-encode any state updates to cut down on traffic, so this is a standard optimisation.

The synthetic workload generator we described in §5.2.1.1 take spawn and despawn intervals as parameters and generates events following these patterns. This is useful for later in this chapter when we investigate how well P2P systems scale as the number players increase. For example, for one version of this workload, we consistently spawn new peers at one minute (game-time) intervals, and despawn players randomly at two minute intervals. This gives us a steadily increasing number of peers at fixed intervals, while

also ensuring we capture the effects of peers despawning (e.g. before they forwarded an update to another peer). This allows us to measure the performance of our system against others for a range of different numbers of peers (to an extreme) without the temporal inconsistency of doing so over a non-synthetic trace. As a net of 1 avatar spawns every two minutes, it does not take long before we have an area of avatars with a density that exceeds the maximum in real traces, thus covering cases of varying extremity.

The RWP mobility model has the evaluation advantage that motion can be very chaotic and cause a high degree of connection churn. This is ideal for stress-testing our system in the worst possible conditions. When measuring the effects of churn, the synthetic trace instead starts with 100 avatars and (de)spawns these at equal fixed intervals such that the total number of peers stays constant. We vary the rate of (de)spawning between every minute (high churn) and every hour (low churn).

Players move in 2000x2000 space of arbitrary units, similar to VON [61]. Fundamentally, this parameter is only important to the extent to which relates to two other parameters: the movement speed of the avatars and the number of avatars. We set the movement speed to be a constant 130 units per second, and we define 1 simulation tick to be equivalent to 1 millisecond. We also set the default update interval to 700 milliseconds, which is the mean we have observed within our "dynamic" trace workload (the "static" trace workload was in the 3.4 second range). This is typical of faster-paced games (see table 2.1). The speed of 130 units per second is the maximum movement speed from our real traces, so we consider it realistic. This gives us similar movement speeds to our traces, in similar sized areas.

5.2.5 Performance metrics used

As we have established in §2.8.1, *consistency/staleness* metrics (which we further deconstruct) are the primary means of comparing the quality of different topologies over the same physical network, we begin our evaluations measuring these for different workloads. We do not make any claims about what the minimum consistency "threshold" required for user satisfaction is. Inconsistency tolerance differs depending on the application — table 2.1 lists such thresholds from the literature across genres.

We focus on three different measures that aim to paint a more complete picture than in literature. Figure 5.4 is a visual aide to help explain these. Here, we focus on the perspective of the dark blue player with the grey AOI. The black players are what the blue player sees, and the green players are their "ground-truth" positions; where the actually are at that point in time. The difference in where we see them and where they

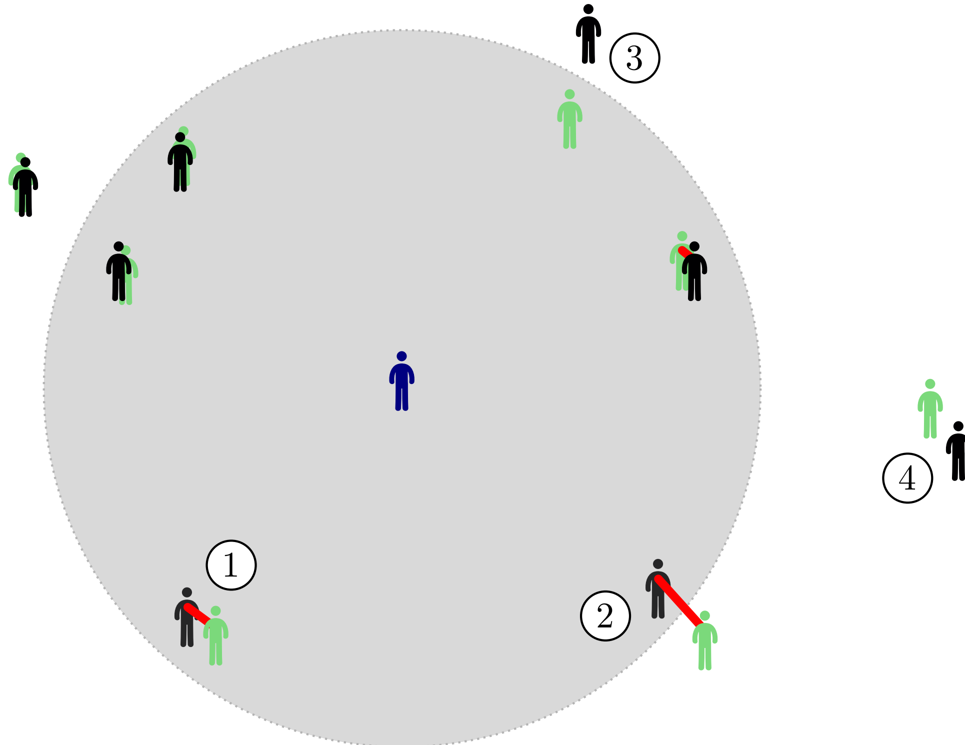


Figure 5.4: Illustration of position drift, missing peers, and extra peers, from the perspective of the dark blue player and their grey AOI

actually are will depend on the update dissemination latency and the motion of the player. For example, the group of three players on the top left are likely idle and therefore close to their true positions.

① is an example of intra-AOI *drift*. Each peer thread, with knowledge of ground truth positions, calculates and stores these drifts for later analysis. In the case of synthetic peers, these are further grouped by total peer count.

② is an example of a situation where our player thinks that a peer is in their AOI, when in reality they have left it. This is an *extra* peer.

Meanwhile ③ is the opposite — a peer's drift is not taken account of because our player thinks that that peer is outside of the AOI when it is not. This is a *missing* peer.

Finally, ④ is an example of a movement entirely outside of our AOI and we therefore do not care about it; the drift distance is not sampled.

Let A be a set of peers that any given player believes to be in their AOI. This set is likely stale/inconsistent, as this player does not receive position updates instantly. In our evaluation environment, players also have instant access to ground-truth position information however, through shared memory. From this, they have G the *actual*, ground-truth AOI set.

- $C = A \cap G$ is the set of peers that a player **c**orrectly presumed is in their AOI
- $E = A \setminus G$ is the set of **e**xtra peers that a player thinks is in their AOI but actually are not
- $M = G \setminus A$ is the set of **m**issing peers that are actually in a player's AOI but that the player thinks are not.

We divide the cardinality of M by the cardinality of G, i.e. $\frac{|M|}{|G|}$, to get the `missingRatio`. This tells us what proportion of the actual AOI is consistent with the peer's view, from zero (completely consistent) to one (completely inconsistent).

We cannot consider this measure on its own however, as it could be that a peer thinks that *all* peers are in its AOI, and the above measure would indicate perfect consistency. We therefore divide the cardinality of E by the cardinality of A, i.e. $\frac{|E|}{|A|}$, to get the `extraRatio`. This tells us how much of a peer's perceived AOI is not part of the ground-truth AOI and is superfluous. For both measures, less indicates higher consistency and is better. We sample these, along with drift distance, at regular intervals of one second.

In plain English, our main general performance are three:

- **Mean Drift Distance** — At any given point in time, a player's known AOI contains a set of peers with potentially stale (position) state. We sample the mean difference between the stale positions and the ground-truth positions of AOI peers throughout the simulation (at one second (game-time) intervals) and calculate a per-peer, trace-wide mean drift distance.
- **Mean Missing Ratio** — The ratio of peers with ground truth positions within a player's AOI but that the player either does not know about, or knows about but incorrectly assumes that they are outside the AOI, averaged across a workload.
- **Mean Extra Ratio** — The ratio of peers that a player assumes is within their AOI but are actually not, averaged across a workload.

In addition to these, we do also explicitly examine more traditional metrics, such as bandwidth and statistics relevant to the experiment in question, and we outline these metrics where pertinent. The consistency metrics described in this section are the most important however, as these and variants of these will appear throughout this chapter.

5.3 Evaluating scalability

The most prominent challenge in developing P2P systems for large-scale NVEs, especially Massively Multiplayer Online Games (MMOGs), is undoubtedly ensuring that these can scale with the number of peers. In order to evaluate the scalability of our system, we consider the most important metrics laid out in §5.2.5. We also compare our method with a range of other representative methods from the literature, which we have implemented for this purpose. This section is split up into two parts. The first looks at performance at scale in general, and the second focuses on scalability in the context of browser limitations.

5.3.1 Performance against AOI density

Recall that we posit that dynamic AOIs are not an adequate solution, as in most use cases, AOI radius is *prescriptive*, meaning that P2P Interest Management (IM) must allow for AOI radii set by the environment, as opposed to the other way around. It follows that AOIs can become very dense, meaning nodes will need to multicast to more peers, which makes scaling difficult. We ask:

How does our system scale with number of peers, compared to others?

We first measure our performance metrics over our traces (static and dynamic) to establish a baseline. As we have no control over how many players there are, we take measurements throughout the entire traces. Our main criticism of related work that attempts these kinds of consistency measurements (see §2.8.1) is that they loose too much information by looking as only averages [33, 61, 70]. Other work treat consistent/inconsistent as binary states as opposed to a spectrum of how consistent/inconsistent [132] (see §2.8.1 and §2.8.7 for more information). To mitigate this, we consider distributions. This allows us to not only see if performance is generally good, but also if there a high variance or an abundance of outliers. This is important as a minority of users experiencing terrible performance at the cost of increasing the average performance is unacceptable.

Figure 5.5 shows distributions of mean drift distance across peers in the form of Tukey Box-Whisker plots. Whiskers show 1.5× inter-quartile range above (respectively below) the third (respectively first) quartile, whereas red diamonds show the mean values. These are overlaid on on top of violin plots to better visualise the nature of the distributions. Topologies with asymmetric peer roles, such as superpeer topologies, are especially susceptible to the main point of criticism we described previously, which can be seen in the

pronounced bimodality in their violins. We will continue to present our results throughout this chapter plotted in the same way as it is a very powerful way of illustrating this kind of data.

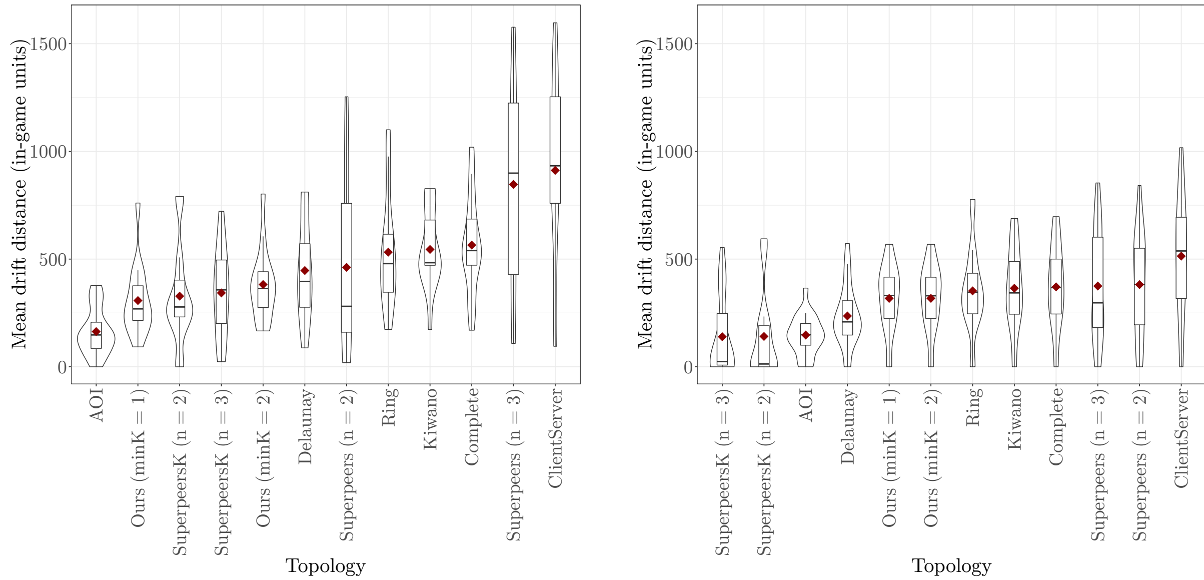


Figure 5.5: Mean drift distance distribution across real environment traces; dynamic (left) and static (right)

We would like to draw attention to a number of observations. First, drift is higher overall in the dynamic area (left) as opposed to the static area (right). This is expected, as a large part of drift is due to topology churn, which we explore in more detail in the following sections. Equally, players are simply more mobile in dynamic areas and therefore their peers will have views of them which are more out of sync.

Across the board, the client-server architecture performs the worst, along with the naive superpeer topologies, and the heavily-connected topologies (Ring, Kiwano, Complete). It is not surprising that naive superpeer topologies are similar to client-server as, under these conditions (no cheaters etc), naive superpeer topologies are almost equivalent to client-server with multiple servers. We notice however that smart superpeer selection can result in networks that outperform ours. These shine in static environments as superpeer churn is much slower.

Similarly, the Delaunay topology is generally on par with ours, and AOI topologies (the most prevalent in literature) have the globally lowest drift spread and outperform ours. Does this condemn our system to inadequacy? No, as this metric does not provide a complete picture as we will see shortly. Equally, this comparison simply served to set the scene under "perfect" conditions. If a superpeer were a cheater for example, or suffered from high loss, this would be devastating to that topology's performance.

As previously alluded to, drift distance alone does not paint a full picture, as peers may be entirely unaware of certain peers that are supposed be in their AOI. Similarly, they may believe that certain peers are in their AOI when they actually are not. This can be linked to the same cause for drift, but can also have algorithmic causes. Figure 5.6 and figure 5.7 show the distributions of *mean missing ratio* and *mean extra ratio* respectively across dynamic and static trace workloads. There are across the entire traces — we found that there were no temporal aspects to these metrics and they are virtually the same across the entire traces.

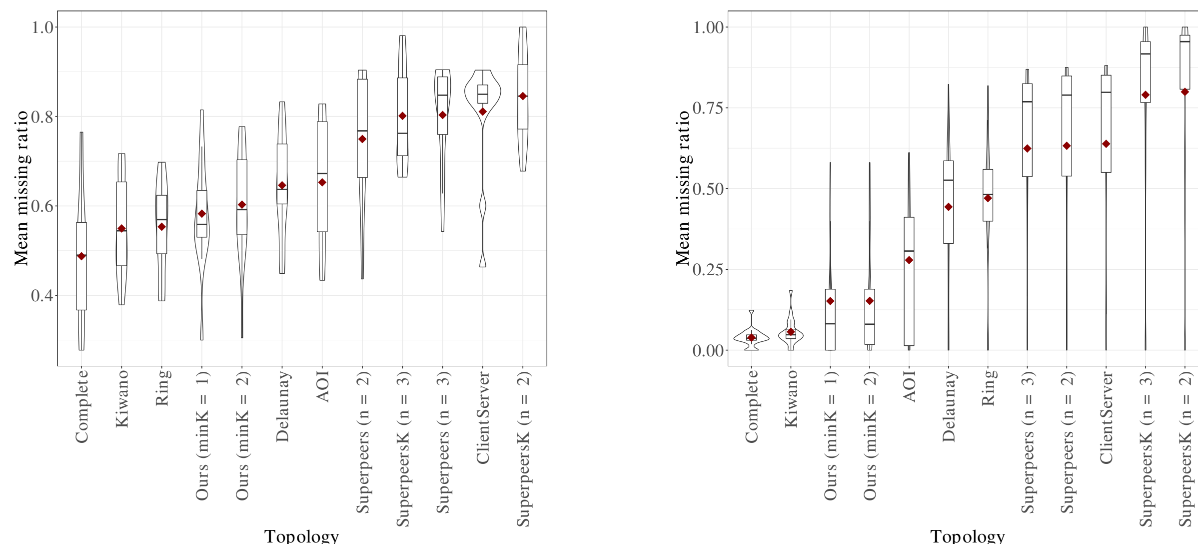


Figure 5.6: Mean missing ratio distribution across real environment traces; dynamic (left) and static (right)

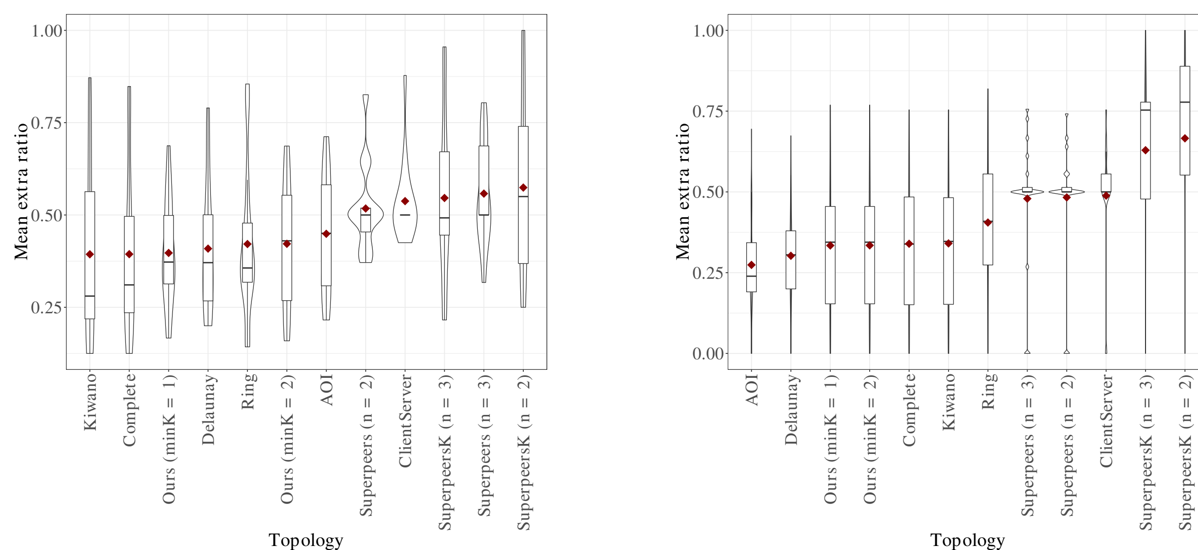


Figure 5.7: Mean extra ratio distribution across real environment traces; dynamic (left) and static (right)

These plots give us a number of insights. When looking at the mean missing ratio of any topology with strong centralisation (client-server and superpeers), we see just how badly they suffer in performance for any environment type. We really cannot stress the importance of these results enough — the previously ideal topologies now miss more peers than not. What is the benefit of maintaining low drift for only a subset of peers? These misses are likely caused by latency along the superpeer links and peers switching between superpeers, since, as expected, static environments (right) are “easier”.

AOI topologies now also perform worse than ours, and suddenly, heavily connected topologies (mainly Kiwano and Complete, but also Ring to an extent) perform very well. This is expected, because part of benefits in the tradeoff of being heavily connected is that you miss far fewer peers due to motion-related churn since they are already connected. One would expect the Complete topology to have a mean missing ratio of zero as, by definition, it is completely connected. However, this is not the case due to inconsistent peers that have just spawned and are in the process of connecting to all other peers. We can see that this penalty is not insignificant in the difference between the mean missing ratio for the Complete topology in the dynamic versus the static environment. While it still preforms the best in the dynamic environment, it is still much worse than the same topology in the static environment.

The mean extra ratio measurements are just as damning for the topologies with centralisation. The only discrepancy here is that the AOI topology performs well in static conditions (mean = 0.2740786, median = 0.2390873, standard deviation = 0.17109030), but not as well in dynamic conditions(mean = 0.4490360, median = 0.4504194, standard deviation = 0.16249360) — almost twice as bad. This is again due to mobility-based churn. In §2, we showed how this topology is the most common in literature for UD. Evaluation workloads are usually static, social environments (such as from Second Life). This result tells us that AOI-based topologies can be acceptable for static environments such as these, but critically, fall short for dynamic environments.

In order to be able to answer the question of scalability properly, we must however go further. We take more detailed measurements over our synthetic workloads that we segment by total peer count. We then take measurements as we incrementally increase the number of peers. This allows us to see how these metrics change as the number of peers increase and extrapolate to extreme cases.

We also note that our synthetic workload is tuned to test for worst case scenarios. Some realistic more realistic scenarios will not necessarily give us any useful insights. For example NVEs where players spend a lot of time on their own will, by definition, have a very high consistency, irrespective of the scalability of the P2P network.

Figure 5.8 shows the relationship between the number of peers and the mean drift distances (locally estimated scatterplot smoothing (LOESS) span 1). Figure 5.9 shows the mean missing ratios and mean extra ratios for context.

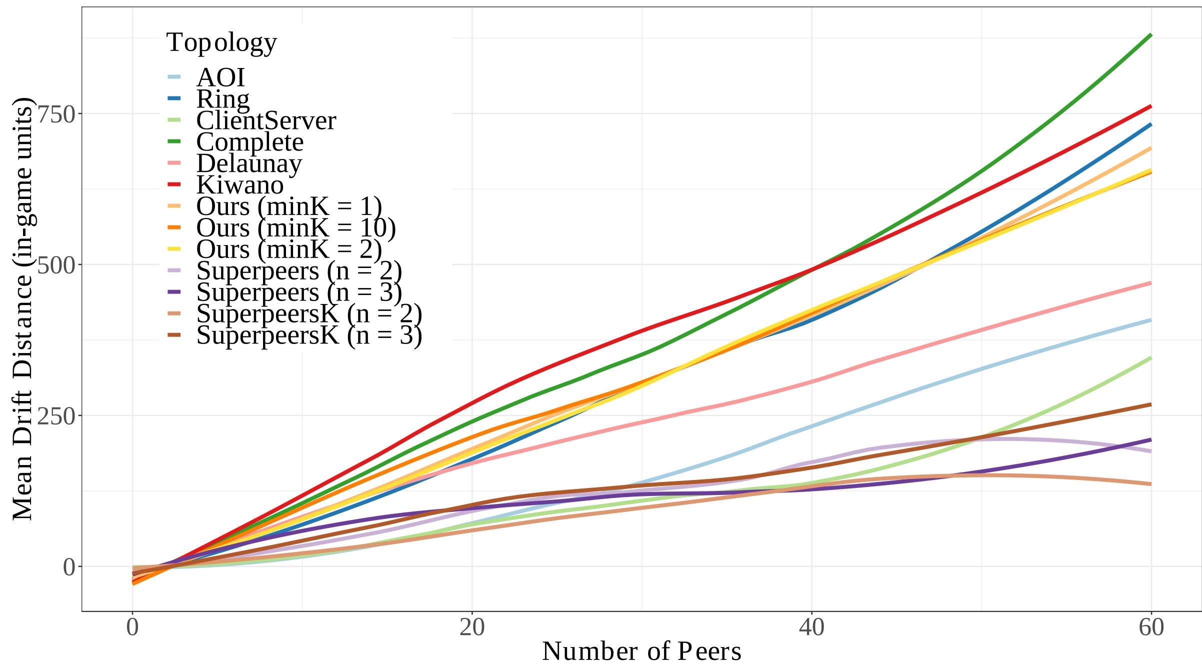


Figure 5.8: Mean mean drift distance across the high-churn synthetic workload for different numbers of players

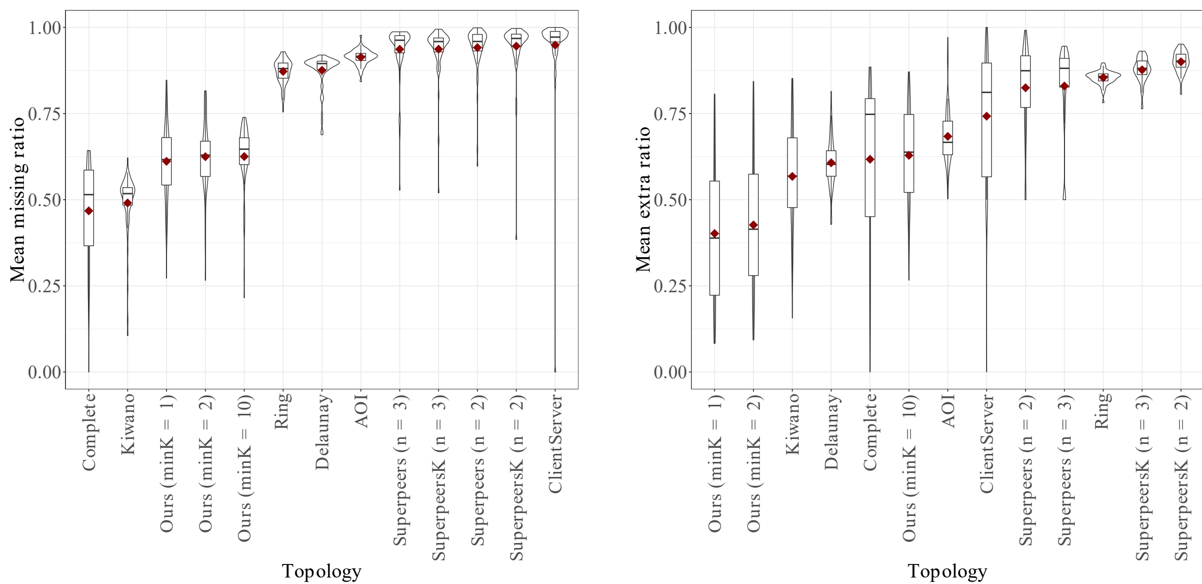


Figure 5.9: Mean missing (left) and extra (right) ratio distributions across the high-churn synthetic workload

First of all, from figure 5.8 we can see that all four superpeer variants, as well as the client-server architecture (the server akin to a superpeer) result in low mean drift distance which increases slowly with respect to the number of peers relative to other topologies. This is followed by Delaunay and AOI topologies, similar to the trace workloads. The mean drift under our system is however lower than the strongly connected topologies (Chrord, Kiwano, Complete) for similar reasons as before.

Again, this on its own does not paint a complete picture. In figure 5.9 we plot distributions of mean missing and extra ratios. We do so with box/violin plots because these are distributions are all but unaffected by the number of players and result in flat lines, so this gives us more information.

Here we can see that, in this high-churn environment, the only reason the topologies that have a lower drift than ours do so is because they entirely unaware of the vast majority of the peers in their AOIs! Our system's mean missing ratio is only surpassed by Kiwano and the Complete topologies, which is specifically because they are so highly connected. Meanwhile our topology's mean extra ratio surpasses even these by a large margin. We can conclude from this that that not only does our system scale well in comparison to other topologies, but it does so under a range of workloads consistently.

5.3.2 Meeting browser constraints

As we have laid out in §3.10, the main reason existing topologies do not fare well in our context, is because of browser constraints. These include a significant overhead in establishing P2P connections, as well as both soft and hard limits to the actual number of connections that a peer can simultaneously maintain. Systems that do not take these into account incur heavy performance costs.

How does per-peer connection count scale for each topology as peer count increases?

Knowing browser connection limits — thanks to our measurements from §3.10 — we can measure the maximum degree (activated edges) for each topology across different numbers of peers to see to what extent these fall below or exceed said limits. We sample every time a topology is computed (i.e. whenever a peer joins/leaves and at fixed intervals). This is over our synthetic RWP workload such that we can control the number of players. We spawn players at one minute intervals and despawn them at deterministically random rate every two minutes. Every sample is then the maximum number of activated edges any one peer has. This means that our player count grows at a rate of one player

every two minutes. Figure 5.10 is a plot of these measurements with the grey envelope corresponding to a 95% confidence margin.

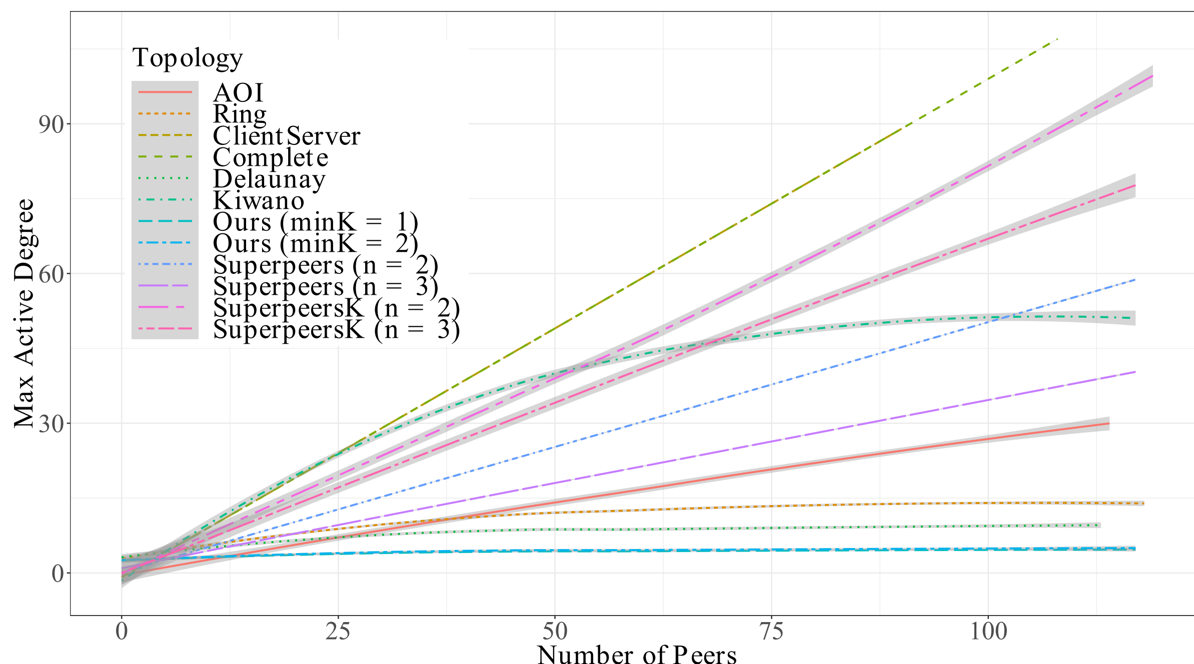


Figure 5.10: Maximum active degree by number of peers over the synthetic workload

Here we make some important observations. First of all, the upper bound is set by the client-server and complete topologies. This is expected, because for the complete topology, all peers are connected to all other peers, and for the client-server topology, the server is connected to all clients — the maximum active degree is the same as just how many clients there are.

Similarly, the maximum active degrees of superpeer architectures have clear, tight linear relationships with the number of peers. The extent of this will of course depend on the number of superpeers. There will always be a more optimal number of superpeers and distribution of peers among them. If we are optimising for number of connections, we can expect the ideal to be a half an half split of superpeers and normal peers, mathematically speaking. If there is only one superpeer, we approximate a client-server model, and if all peers are superpeers then we are back at the Complete topology, unless the connections between superpeers adopt fat-tree-like topology for example.

The difference between two and three superpeers is clear here however, as three superpeers always result in a less steep correlation. We also note that the superpeer topologies that perform better (which use clustering for superpeer selection and peer allocation) have a higher maximum active degree and are also the most unpredictable. This is because the peer distribution for these is not symmetrical and highly dependent on virtual

position, while naive superpeer topologies will split peers among superpeers equally in a round-robin fashion.

AOI-based topologies, which are the most common in literature, seem quite attractive and "obvious" solutions at first, but they can have their flaws. Here we see that the maximum active degree increases linearly with player count, however this is misleading. AOI maximum active degree increases with AOI *density*, not player count. Since we are spawning players in a fixed-sized area, naturally the density of the AOIs will increase, but we have already shown in §3.5 and other parts of §3 that there is a natural "human" limit to how dense AOIs get — the human brain is simply not capable of interacting with that many people at the same time. Players tend to space themselves out and leave when an area gets overcrowded. It is difficult to have any meaningful interaction when avatars are literally on top of each other, and indeed the physics engines of many NVEs preclude that possibility. That being said, there is nothing stopping this topology from easily hitting browser connection limits with a large enough AOI in a sufficiently popular area.

The remaining topologies do not show a linear relationship. Kiwano, based on Delaunay power graphs, has a significantly higher connectivity than plain Delaunay, but still maintains many of its properties, making it quite scalable for this particular constraint. Ring is, of course, $\log 2$ as expected, and a Delaunay graph is even more efficient.

The simplest form of our topology reduces to an Minimum Spanning Tree (MST). This is not necessarily the topology with the lowest maximum active degree (that would be one where nodes are literally chained together such that the maximum number of active neighbours a chain can have is two). Our topology can get very close to this however and scales extremely efficiently. We can comfortably scale to unrealistic numbers of peers and still fall well within browser bounds. This holds true even when we add our redundant connections by increasing the connectivity parameter; the difference is negligible, which is very promising.

This is amplified by the fact that we maintain a control channel with the signalling server, which tells us when peers disconnect, and allows peers to remove dead connections properly (which can be a problem as we have have shown in §3.10. If they are not, too high a churn will cause these limits to be rapidly exceeded.

This is of course not the only constraint in a browser context. As we have alluded to earlier, user devices and internet connections also impose limits. A P2P solution to game networking that also increases peers' bandwidth usage by orders of magnitude is not practical, especially if peers simply do not have the capacity. This is largely due to Application-Layer Multicast (ALM) causing a disproportional increase upload

rates, and unfortunately download speeds/capacity tends to be prioritised over upload speed/capacity with modern consumer connections.

How are upload/download rates affected as peer count increases?

We measure the upload and download rates for each peer and plot these distributions as box/violin plots in figures 5.11 and 5.12 for the dynamic and static workloads respectively. To measure these we had each peer (or more accurately, peer session) keep track of exactly how many bytes it uploads and downloads, and we divide this by the time the lifetime of the session in order to get per-peer averages. We opted not to limit the y axis in order to illustrate the extent of the extremes, which tells us much more, however the x axes (topologies) are ordered by mean for clarity.

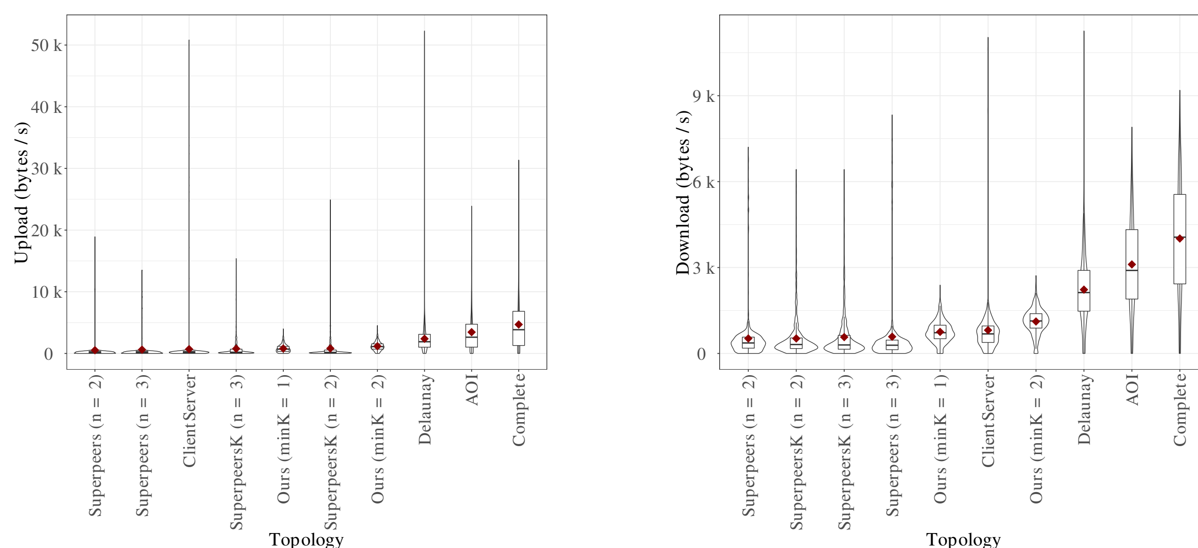


Figure 5.11: Mean upload (left) and download (right) rate distributions across the dynamic environment workload ordered by mean

For the most part, while the overall bandwidth requirements differ across workloads (as expected) the patterns remain the same. As expected for ALM, download rates are much lower upload rates across the board, owing to loss etc.

We can immediately see that a Complete topology will have the highest requirements, followed by AOI and Delaunay. This is because for these topologies, an AOIcast of one byte to n peers will result in an upload of n bytes, or (in the case of Delaunay) approaching n bytes due to high connectivity.

Note that Kiwano and Ring are missing from these plots. Both have upload and download rates so high, that the other topologies are completely flattened in comparison. Besides our topology, Ring is the one system where peers can forward outside of their

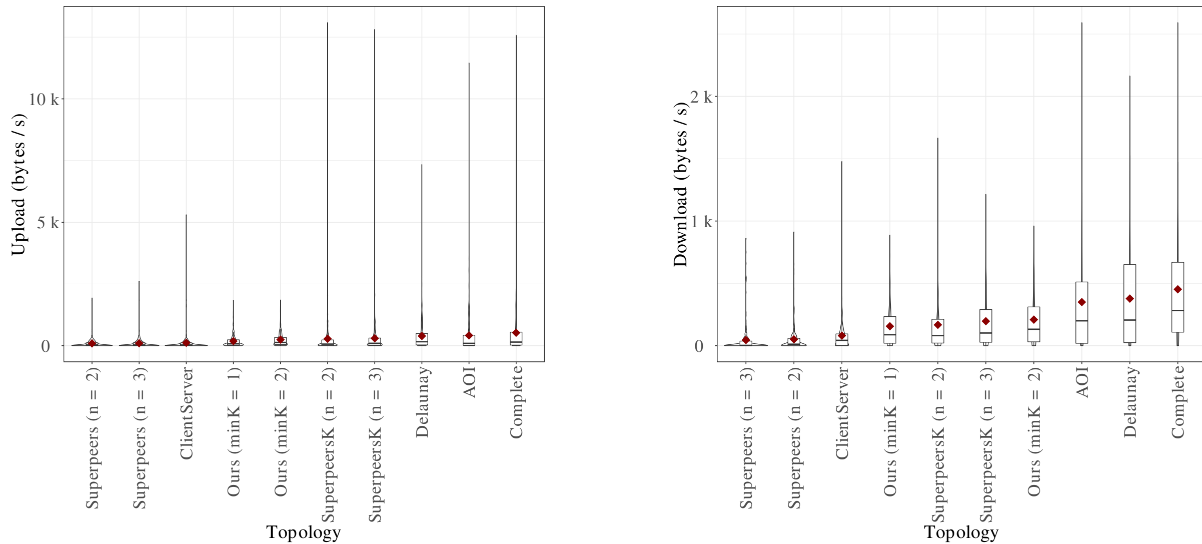


Figure 5.12: Mean upload (left) and download (right) rate distributions across the static environment workload ordered by mean

AOI (to non-superpeers of course). Recall that we limit the number of hops an update can make for this reason. Kiwano is topologically very strongly connected, which makes is akin to a Complete topology that supports forwarding. For similar reasons, Delaunay topologies have high bandwidth requirements, however these are still reasonable when compared to Kiwano.

On the other hand, client-server and superpeer topologies will naturally have lower bandwidth requirements on average, as peers may only need to send/receive an update to/from a single server or superpeer, however the trade-off is that the server or superpeer will need to have a significantly higher bandwidth capacity. This creates the disproportionately high upper bounds where a few nodes take the majority of the load.

The most important takeaway from these plots is that our system while, of course, not able to beat centralised architectures on average, achieves consistently low upper bounds for bandwidth requirements across peers. This is especially important for browser and mobile use cases.

For good measure, we also examine how bandwidth usage scales against number the number of online peers under our synthetic workload where we can control that manually. Figure 5.13 shows the relationship between these two for upload (top) and download (bottom) rates.

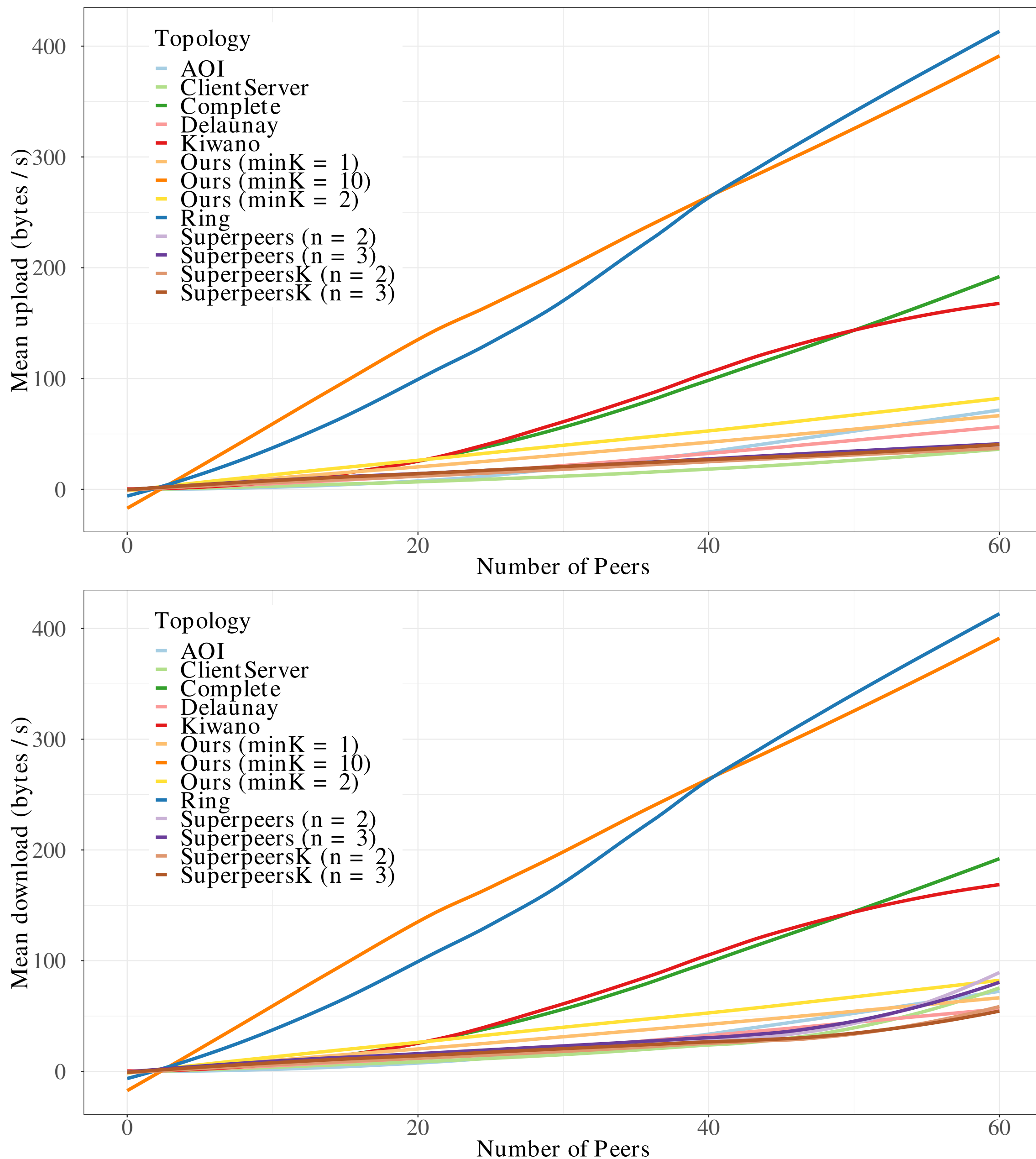


Figure 5.13: Mean upload (top) and download (bottom) rates across the synthetic workload for different numbers of players

These are of course less realistic, and we can see that the upload rate plots and the download rate plots are virtually identical. These figures mainly serve to illustrate that as the number of peers go up, our topologies, despite requiring forwarding outside of AOIs, scale slowly and near-linearly with the number of peers, while others do not.

In order to include Kiwano and Ring on this plot, we lowered the global maximum hop limit to 3 hops, which shows that Kiwano behaves similarly to a Complete topology. Ring continues to have the highest requirements.

Recall that the only two systems that can forward messages outside of their AOI are Ring and Ours. For illustration, we tested a variant of our topology with a minimum connectivity variable of 10, such that it can be comparable to Ring in connectivity. This variant turns out to have similar requirements as we can see here. This simply serves to illustrate that our system is easily adjustable and versatile depending on developer requirements and the use case in question.

Ultimately, these measurements tell us that, for typical workloads, the bandwidth requirements under any topology fall well bellow typical browser constraints. At scale, the topologies that previously performed better on mean missing ratio, Kiwano and Complete, have over double the bandwidth usage of Ours. This means that these will reach bandwidth limits at least twice as fast as ours; at half the number of players. In such a restrictive context, Our topology is the best choice.

5.4 Evaluating churn sensitivity

How sensitive is our system to peer/connection churn, compared to others?

In §3.10 we highlighted one of the primary browser constraints to be the time it takes to establish new connections. Our first solution to this is to pre-connect to peers outside of the AOI such that by the time they enter the AOI, the connection is already well established. A secondary solution is to modify semantic distances to account for longevity of a connection, such that longer-lived connections are preferred. This effectively applies a switching cost and avoids incurring these overheads.

To test how well this performs, we vary topology churn and examine how many updates are lost as a result of connections not having had enough time to be established (which we call the connection *cooldown*; see §5.2.2 for more details).

We use the real cooldown measurements (from §3.10) and iPlane measurements (from §5.2.2) to deterministically apply a cooldown (which is a function of link latency) to each newly established connection. We then count the number of updates that are dropped as a result of this as well as due to despawning.

For context, we also show how many updates are lost in this way over the trace workloads. For these, we have no control over the churn, unlike with our synthetic workload. In the latter case, we segment the data by the degree of set "churn". We define "churn" as a value between 0.0 and 1.0 which maps to (de)spawn intervals between one

hour and one minute. As the spawn and despawn rates are the same, the number of peers remains constant. We spawn 30 peers immediately in the beginning.

Figure 5.14 shows the proportion of updates lost due to cooldown against the total sent successfully, across trace workloads. The sum of these is the total number of updates attempted to send (no other loss).

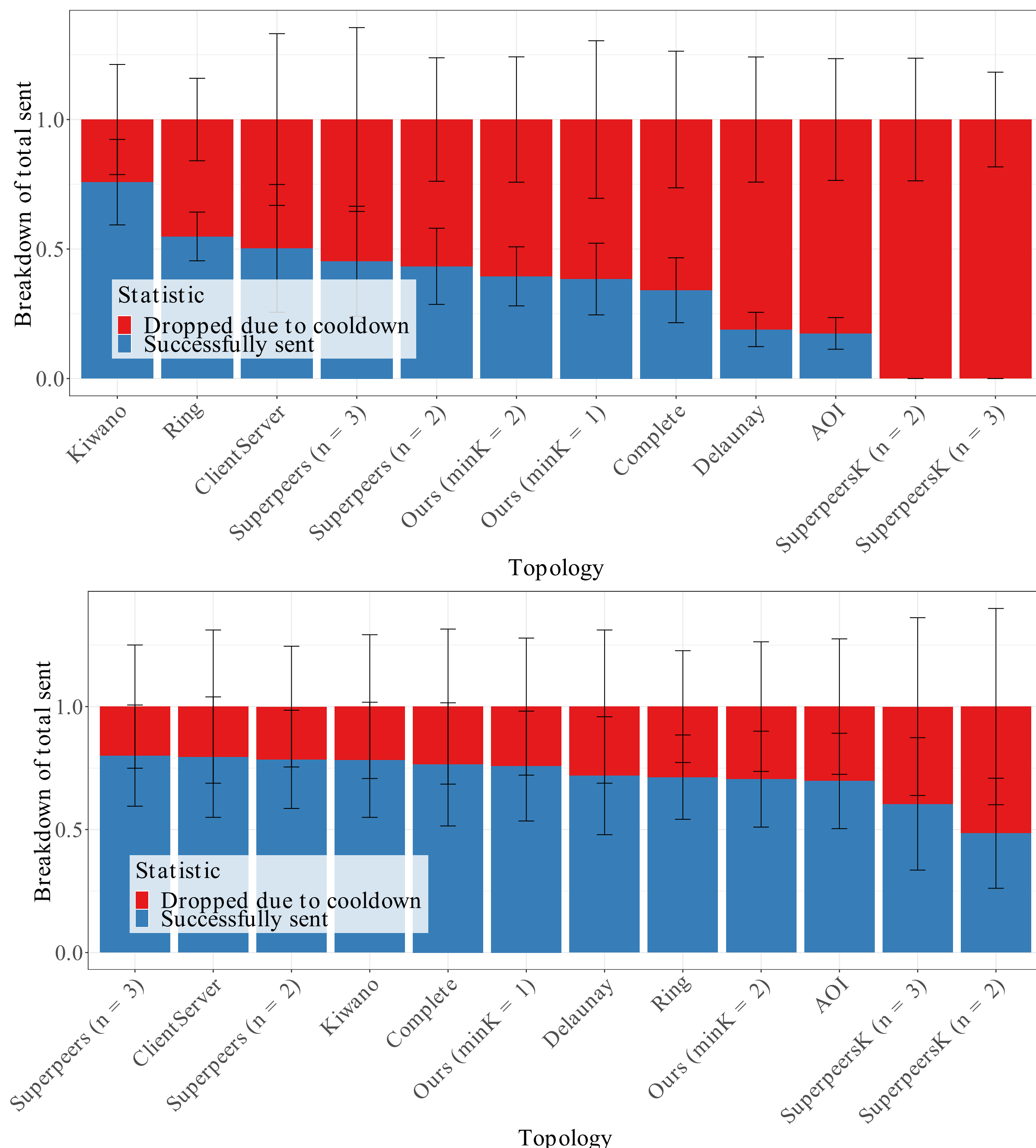


Figure 5.14: Proportion of updates lost due to cooldown across dynamic (top) and static (bottom) workloads

There is a lot to unpack from these figures. We can see that many topologies lose a large portion of data to cooldown. This confirms just how important a constraint connection cooldown is for a browser context. We would like to note however that these measurements do not paint the full picture and consistency metrics are much more important. For example, our system pre-connects peers to circumvent the cooldown penalty, however this will only result in *more* updates being dropped as a peer attempts to send it down a link that is not yet ready, while in actual fact we achieve higher consistency. Nonetheless, this is important to consider because it tells us how susceptible a topology is to the cooldown overhead and therefore churn.

Recall that the dynamic environment naturally has higher churn, however even in the static environment, "intelligent" superpeer selection can result in high loss due to switching superpeers too frequently. There are strategies to get around this, for example by limiting the rate at which superpeers can change — the "superpeer churn" — but this is outside of our scope. It goes to show that this class of topology needs much more nuanced optimisations or else risk total loss, like under the dynamic workload.

Topologies with low rates of connection/disconnection, like naive superpeer topologies and client-server architectures lose fewer updates to cooldown, regardless of motion. This is the one advantage to having a topology that is independent of virtual position and mobility, akin to strongly connected topologies such as Complete and Kiwano. Our topologies on their own create many situations where links are not ready for sending, but as we have shown, we overcome this entirely through pre-connecting peers, and as a result, our system is only affected by churn caused by spawning/despawning as opposed to motion.

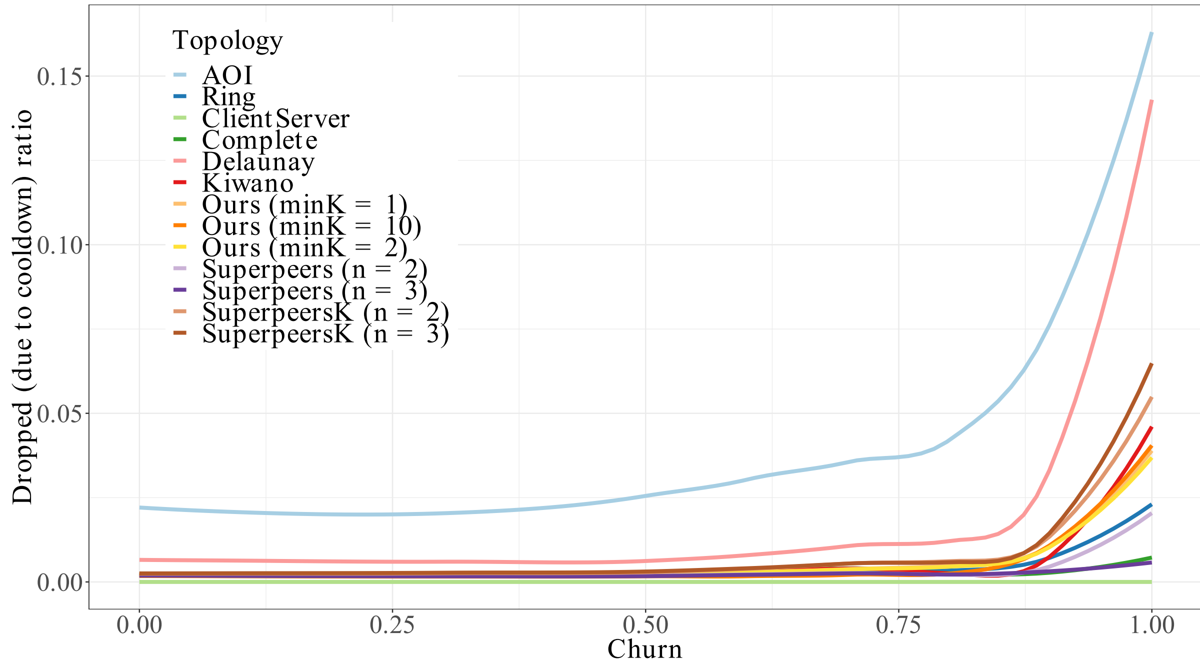


Figure 5.15: Mean ratio of updates dropped due to cooldown over the synthetic workload for different levels of churn

Figure 5.15 shows the effects of spawn/despawn churn as we attenuate it on top of our high mobility churn synthetic workload (LOESS span 0.6 for clarity in trend). In this environment, connection changes due to churn caused by mobility dominate over those caused by spawning/despawning. For the most part, we can see that the topologies stack up against each other similar to under trace workloads. However, we can see that overall, effects of spawn/despawn churn are limited until we reach a churn faster than approximately seven minutes. Below that point, the number of packets lost due to cooldown increases exponentially.

Besides the usual suspects that remain unaffected by churn due to their different architecture, our approach is the least affected by churn, regardless of the level of connectivity. This simply mirrors the trace workload results but under more controlled conditions, proving our topology resilient to churn under a range of workloads with a lot of variation.

In summary, we can see that building a system that is insensitive to topology churn by design pays a lot of dividends in a deployment environment that severely penalises topology churn (the browser). We have shown that we were able to mitigate the effects of topology churn effectively, despite our topology being extremely sparse.

5.5 Evaluating packet loss resilience

As P2P networks can be significantly more unreliable than client-server networks, a common evaluation in literature (see §2.8.6) is reliability. Similarly, we measure the effects of a globally defined packet loss ratio for all inter-peer links. Recall that UD is in practice over lossy/unreliable protocols like UDP, and retransmission is known to add no value as these updates are short-lived — by the time they are retransmitted, they are already irrelevant. Because of this, it is important to quantify the effects of packet loss on NVE performance metrics.

In this section we vary the loss ratio through which each link has a deterministically random probability of dropping any given update. Lost updates means more inconsistent information across peers, so in order to determine the effects of packet loss on performance, we measure the mean drift distance as the loss ratio is increased, in order to answer the question:

How resilient is our system's performance to packet loss, compared to others?

Other consistency metrics do not tell us anything here as they are unaffected by loss. In our context, the signalling server is what tells players which peers to connect to. While this is strongly dependent on the way topologies are computed, we are focusing specifically on loss between peers. The connection to the signalling server is a reliable WebSocket connection and the reliability of this connection is out of scope of our evaluation. We therefore focus specifically on drift distance, and not missing/extra peers, as the latter is unrelated to inter-peer packet loss.

We run this experiment using our synthetic workload (high churn) in order to focus on the worst case effects. The loss ratio remains constant for an entire run, and we repeat the whole experiment at different loss ratios from 0.0 (no loss) to 1.0 (total loss) at 0.1 increments. The signalling server tells players peers' initial spawn point, so these measurements are on subsequent updates. Figure 5.16 shows how each topology performs as we increase the loss ratio.

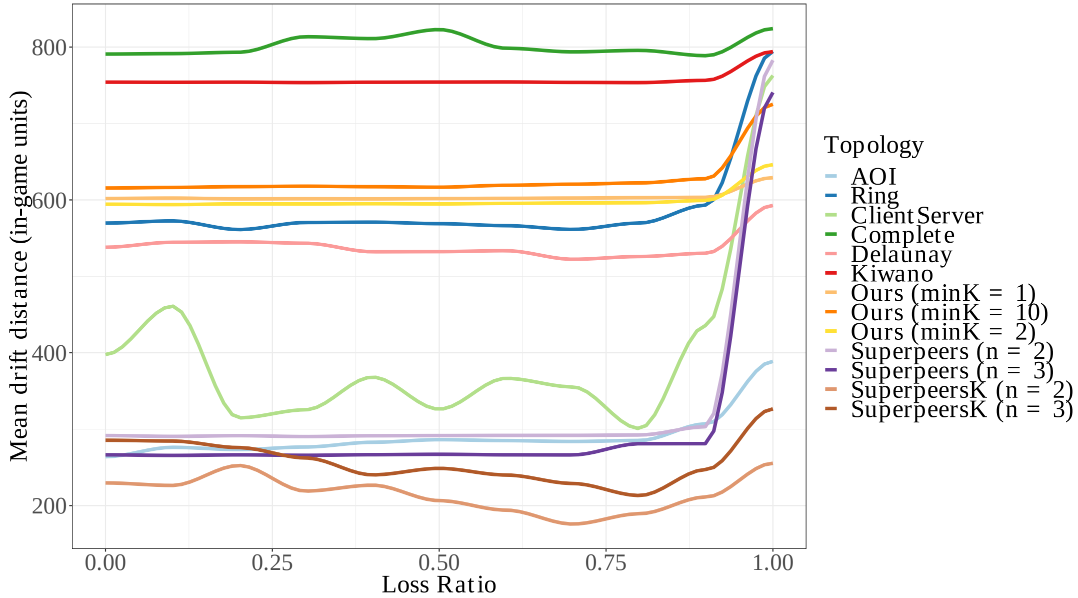


Figure 5.16: Mean drift distance under a range of artificially induced loss ratios

When interpreting these results, we must keep certain caveats in mind. The drift distance for example is naturally bounded by the size of the environment. Similarly, it is affected strongly by mobility. For example, in the case where all packets are lost, players will have no conception of where their peers are beyond what the signalling server tells them at the start. In this case, and divergence from the spawn point counts as drift, but a player can linger close to that point, or leave and come back again. “A broken clock is correct twice a day” as they say.

From figure 5.16, we make a number of important observations. The first is that, while client-server and superpeer topologies can result in low drift overall (as we have also seen in §5.3) these are the most sensitive to loss, especially at the higher ratios. This is expected, because normal nodes do not perform ALM for these, but instead unicast to a supernode or server which then “amplifies” this update. If that update is dropped, then nothing is forwarded.

Meanwhile, while densely connected topologies (Complete, Kiwano) can perform badly over all due to updates not taking the most optimal routes (see §4.4), they are least affected by loss due to the high redundancy and replication of updates. Since these topologies have relatively high drift regardless, they reach the natural upper bound so it seems like they are less affected that they actually are.

Our topologies on the other hand perform well and are least affected by loss, even when their connectivity parameter is set to the lowest values possible. This tells us that the trade-off in sparseness is worthwhile, even in networks that may suffer from high loss.

Some topologies respond exceedingly unpredictably to packet loss, most notably the ClientServer topology, despite the fact that these are means over a large number of samples. With shorter paths especially, such as in the ClientServer case, there are fewer opportunities for an update to be dropped. This creates a lot more extremes and there are fewer AOI peers in an update's path that can be updated as it passes them. These extremes create much more chaotic drift under a workload with turbulent motion, even though both motion and loss are deterministic and repeated identically for every tested loss ratio.

We opted not to include confidence ribbons around these means as they occlude the lines and make the plot hard to read, however we would also like to note that the topologies where the lines are less regular had a very large spread in drift distance (i.e. some peers experienced high drift and others did not). This is expected for heterogeneous network roles such as superpeer networks. While our experiments are deterministic and repeated exactly the same way under each loss ratio, that tiny change can have significant effects on the performance, as one flap of a butterfly's wing can mean a different superpeer gets selected.

This variance is also a major downside for game networks, where a superpeer may have an unfair advantage by having less stale/inconsistent information, simply because they have more connections. This is a problem that many implementations do not consider, and the only way to solve this (all else being equal) is to artificially induce lag towards superpeers or other topologically advantaged peers, which can create a myriad of other complications. Our networks do not suffer from this problem.

5.6 Evaluating cheating mitigation

A common theme throughout this thesis has been the need to tackle the problem of bad behaviour in a P2P NVEs. In §2.5.2 we clearly laid out the challenges that exist in a P2P setting that do not exist in a traditional client-server setting. Then, throughout our work, we further narrowed down exactly which constraints a system would need to meet in order to effectively mitigate cheating. In this section, we aim to quantify cheater influence and its effect on performance across topologies, and demonstrate how our system effectively mitigates cheating.

5.6.1 The effectiveness of limiting cheater influence

At the topological layer, a system that effectively mitigates cheating should intuitively cause cheating nodes to be less connected. Our system attempts to do this, while introducing design features that flat out make certain kinds of cheating impossible. To understand how well different approaches do this, we first ask:

How influential are cheaters in our system, versus others, at different proportions?

A important caveat to note before we address this question through experimental evaluation is that our system prevents forwarding nodes from tampering with updates by design (see §4.7). However, it is still important to avoid cheaters from being too connected as they can affect the network in other ways, for example by intentionally dropping the updates of other peers.

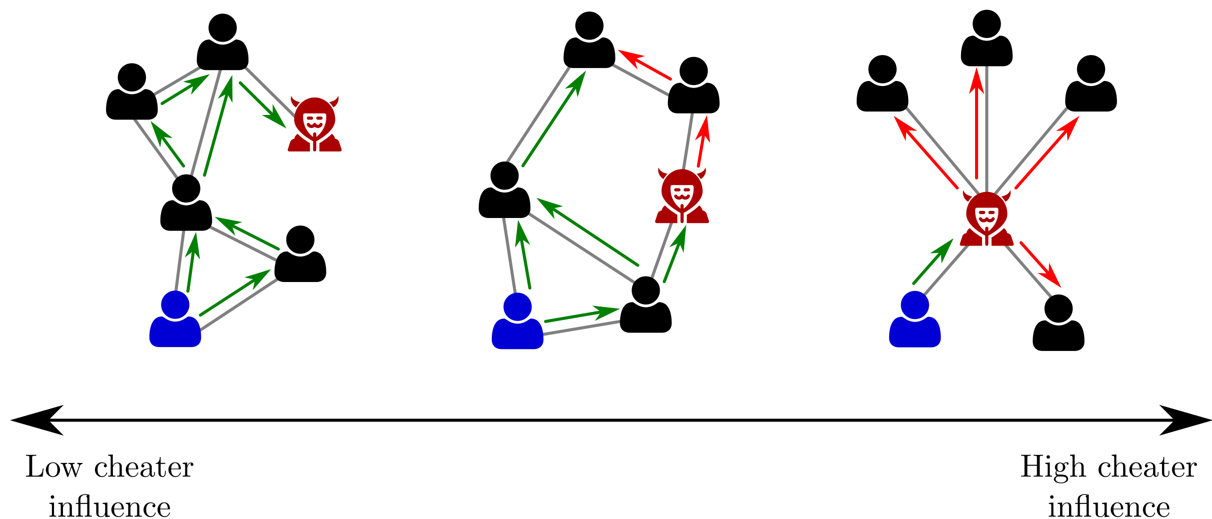


Figure 5.17: Illustration of cheater peer (red) influence on the propagation of updates from a source peer (blue) in networks across a spectrum from low to high cheater influence

Figure 5.17 depicts cheater influence in an abstract sense. On the right side of the spectrum, we have have a network where updates sent from the blue peer have no choice but to take a path through a cheating peer (red) to reach other peers. In the middle, we have a network with alternate paths around the cheater, lowering their influence. In this example, the topmost peer receives the same update via two different connections, the redundancy of which allows them to negate the cheater's influence even further. On the far left, the topology makes the cheater a terminal node, minimising their influence.

To address the question of cheater influence, we measure the normalised betweenness centrality [37], a standard and well-established node-wise metric based on the number of shortest paths that pass through a node, widely used to measure node-wise influence in a network. This is critical here because the non-shortest paths are not relevant — if a player receives an update via a faster path, and the same update again via a slower one, the one that came later is ignored anyway. The *mean normalised cheater betweenness* (MNCB) is an average across all cheating players every time the signalling server recomputes the topology.

$$
\begin{aligned}
g(v) &= \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \\
\text{normal}(g(v)) &= \frac{g(v) - \min(g)}{\max(g) - \min(g)} \\
\text{MNCB} &= \frac{1}{|C|} \sum_{c \in C} \text{normal}(g(c))
\end{aligned}
\tag{5.1}
$$

We express this metric more formally through equation 5.1[5]. Here, the betweenness centrality for node v is given by $g(v)$. σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v. As this value scales with the number of nodes, it must be normalised to become independent of size ($g \in [0, 1]$) as our networks have variable sizes. Finally, we sum the normalised betweenness centrality across the subset of nodes that are marked as cheaters (denoted by the set C) and divide this by the cardinality of that subset to get a mean.

In our evaluation setup, the (deterministic) probability of a node being a cheater is controlled globally. We repeat the entire experiment (with both static and dynamic trace workloads) under a range of cheater probabilities, from 0.0 (no cheaters) to 1.0 (all cheaters) in increments of 0.1. Players of course have no indication that a peer is a cheater, however our evaluation framework does track how many times an update is forwarded by a cheater, how many times a player accepts a "corrupted" update (one that passed one or more cheaters on the way), as well as the corruption "magnitude" across these accepted updates.

We must note that when comparing topologies here, it is not necessarily apples to apples. First of all, not all systems support update forwarding. The *Complete* and *AOI* topologies have no need for this as they aim to have updates delivered in a single "unicast" hop. Equally, we assume that the server in the *ClientServer* architecture is secure and

[5]The notation here is based off of Wikipedia's formalisation of betweenness centrality: `https://en.wikipedia.org/wiki/Betweenness_centrality`

cannot be a "cheater". Some topologies are also disconnected by design and do not allow broadcasting to all peers, only trying to connect AOIs, which already mitigates the influence of cheaters at the cost of a feature. We stress that our system does not just connect AOIs, but all peers and allows broadcasting.

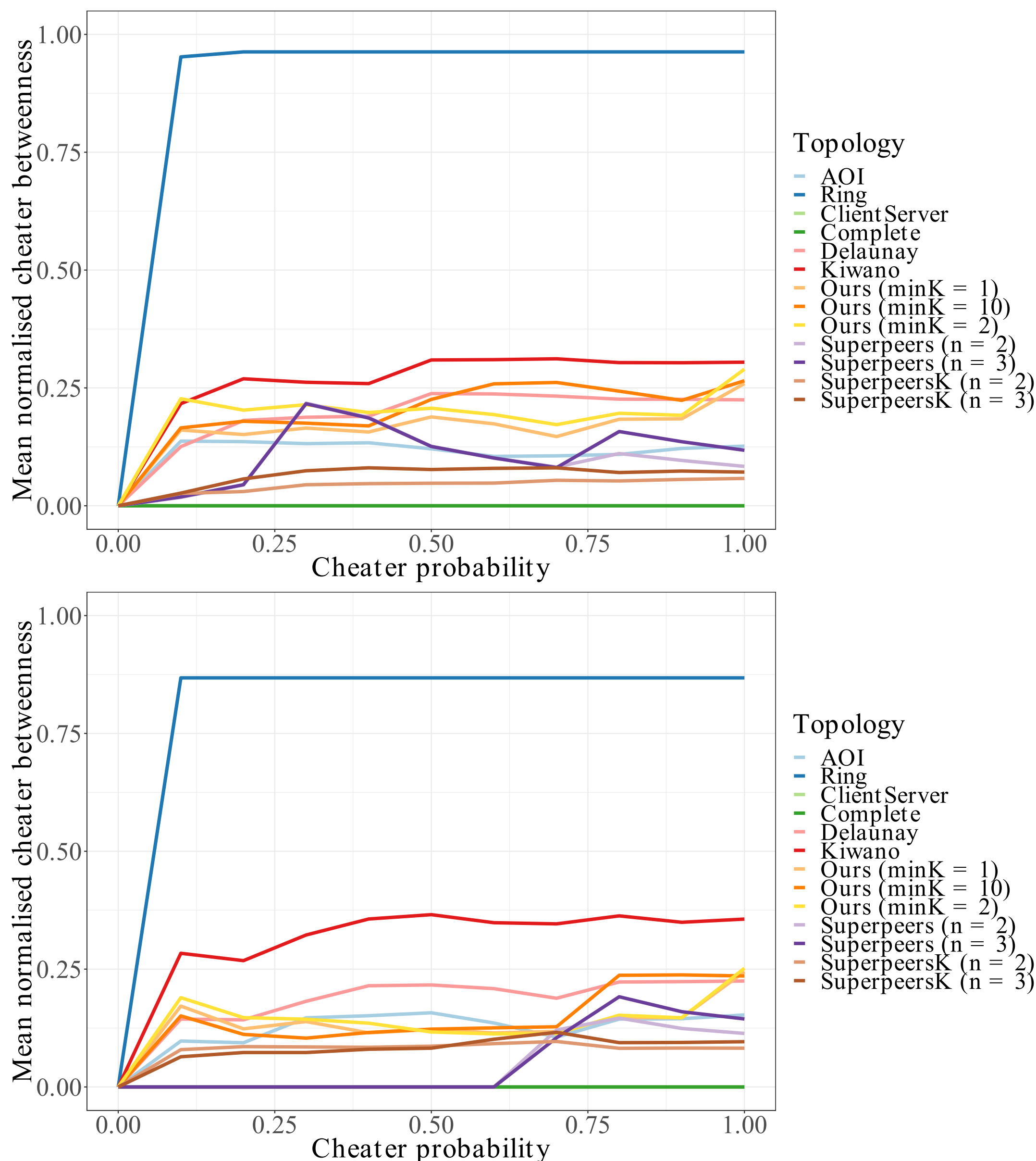


Figure 5.18: Mean normalised cheater betweenness measures for different cheater probabilities across the dynamic (top) and static (bottom) workloads

Secondly, cheaters can cheat at different "magnitudes". For example, one cheater may cheat by only occasionally slowing down forwarded updates slightly while another may drop these outright in bursts. Since we are more interested in measuring different topologies in relation to each other and the proportion of cheaters, we freeze this variable

and make it constant for all cheaters. Specifically, we make these "protocol-level" cheaters (the most pertinent in this context; see §2.5.2) and have all cheaters simply slow down all updates they forward to effectively double the expected latency of link they are about to forward it down, making cheaters "choke-points" in the network. We run this experiment to understand the influence of these choke-points and plot the results in figure 5.18. Normally we would add 0.95 confidence ribbons to this plot, however the variance here can be quite large, making the plots difficult to read, so we instead separated this information out into tables 5.2 and 5.3.

Table 5.2: Table of standard deviations of mean normalised cheater betweenness measures for every topology (rows) and cheater probability (columns) across the dynamic workload

Topology	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1**
AOI	0	0.231	0.203	0.195	0.195	0.154	0.138	0.115	0.084	0.081	0.084
Ring	0	0.214	0.189	0.189	0.189	0.189	0.189	0.189	0.189	0.189	0.189
ClientServer	0	0	0	0	0	0	0	0	0	0	0
Complete	0	0	0	0	0	0	0	0	0	0	0
Delaunay	0	0.138	0.191	0.159	0.149	0.130	0.119	0.112	0.094	0.087	0.087
Kiwano	0	0.317	0.344	0.278	0.271	0.259	0.257	0.248	0.251	0.244	0.241
Ours (minK = 1)	0	0.243	0.181	0.170	0.150	0.133	0.106	0.082	0.076	0.067	0.060
Ours (minK = 2)	0	0.243	0.182	0.184	0.156	0.138	0.133	0.126	0.099	0.095	0.092
Ours (minK = 10)	0	0.138	0.125	0.139	0.137	0.139	0.146	0.155	0.149	0.148	0.154
Superpeers (n = 2)	0	0.123	0.179	0.165	0.145	0.135	0.124	0.119	0.093	0.073	0.064
Superpeers (n = 3)	0	0.131	0.184	0.169	0.148	0.140	0.131	0.121	0.123	0.095	0.081
SuperpeersK (n = 2)	0	0.077	0.070	0.078	0.070	0.062	0.059	0.061	0.048	0.049	0.048
SuperpeersK (n = 3)	0	0.105	0.109	0.102	0.103	0.083	0.089	0.086	0.063	0.062	0.058

Table 5.3: Table of standard deviations of mean normalised cheater betweenness measures for every topology (rows) and cheater probability (columns) across the static workload

Topology	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1**
AOI	0	0.174	0.136	0.135	0.100	0.096	0.083	0.065	0.063	0.063	0.063
Ring	0	0.342	0.342	0.342	0.342	0.342	0.342	0.342	0.342	0.342	0.342
ClientServer	0	0	0	0	0	0	0	0	0	0	0
Complete	0	0	0	0	0	0	0	0	0	0	0
Delaunay	0	0.178	0.150	0.146	0.179	0.171	0.174	0.116	0.116	0.117	0.115
Kiwano	0	0.303	0.271	0.284	0.306	0.306	0.294	0.291	0.300	0.297	0.302
Ours (minK = 1)	0	0.256	0.190	0.183	0.126	0.122	0.120	0.081	0.064	0.080	0.080
Ours (minK = 2)	0	0.181	0.155	0.144	0.090	0.084	0.084	0.069	0.086	0.096	0.093
Ours (minK = 10)	0	0.173	0.105	0.107	0.135	0.137	0.145	0.157	0.186	0.198	0.189
Superpeers (n = 2)	0	0	0	0	0	0	0	0.146	0.099	0.095	0.093
Superpeers (n = 3)	0	0	0	0	0	0	0	0.094	0.116	0.102	0.097
SuperpeersK (n = 2)	0	0.130	0.115	0.105	0.106	0.092	0.100	0.081	0.061	0.071	0.071
SuperpeersK (n = 3)	0	0.168	0.138	0.138	0.135	0.125	0.142	0.156	0.096	0.096	0.095

Here, the *Complete* and *ClientServer* topologies are of course entirely unaffected by any amount of cheaters. The *Complete* topology because any peer can reach any other peer in just one hop, and the *ClientServer* topology because all forwarding is done by the server which cannot be a cheating player by definition (unlike in superpeer architectures).

We can also see that the patterns remain largely the same across area types (dynamic and static). Indeed these are immune to changing the cheating "magnitude" per cheater. The only major discrepancy are the naive superpeer topologies. This is because the point at which cheaters have a large influence is largely down to luck — if a cheater is selected as a superpeer (which is much more likely as there are more and more cheaters) this will affect the topology much more significantly. Otherwise, they are essentially equivalent to a multi-server client-server architecture.

Notice also that intelligent superpeer selection is more robust against cheaters. This is because, like our method, network metrics are taken into account. The superpeer selection is resource-based and cannot tell the difference between peers that drop or delay packets intentionally versus peers with a genuinely bad connection. This makes it so that cheaters are less likely to be selected as superpeers.

Our method's resilience to cheaters is much more subtle on the other hand. One would expect extremely sparse networks such as ours to be *more* susceptible to the influence of bad actors than others, as sparsity is generally inversely proportional to both edge and vertex connectivity. The reason this is not the case is twofold. First, our neighbour selection by design deprioritises cheaters and bad connections (see §4.5). Second, we can easily control the minimum connectivity of our networks, creating alternate "safe" paths in the network using our $minK$ parameter. If we set this to the maximum, our networks would become completely connected and be entirely unaffected by protocol-layer cheating.

We can also see that one of *Ring*'s desirable properties (the high connectivity and short diameters) makes it almost guaranteed that paths will cross cheaters after a certain point. It is possible that if its hashing were location-based (i.e. players closer together in the virtual space were also closer together on the ring) that this problem would be less prominent.

The AOI topology is generally on par with our methods, however cannot be adjusted, and this resilience comes at a cost to other performance metrics and features (such as the ability to broadcast across all peers). It is essentially a lesser form of a *Complete* topology so will naturally have similar properties.

Finally, we know that *Kiwano* networks, like *Ring*, tend to have very high connectivity, however unlike a *Complete* topology, peers are not all necessarily a single hop away from

each other. This explains the MNBC rising to 0.3 (dynamic) and 0.37 (static) as for *Kiwano*, as it can give cheating nodes a high level of influence, which becomes especially dangerous when augmenting a system with some sort of peer voting mechanisms.

Thus, we have demonstrate that, despite sparsity, our system limits the influence that cheaters can have. In addition to this, our system allows developers the versatility to adjust the neighbour selection metric weights to prioritise cheater mitigation even further, at the cost of other dimensions of performance, in an application that is particularly susceptible to cheaters/cheating.

5.6.2 Cheater impact on performance

This previous results alone do not paint the full picture. We are interested in how cheating affects NVEs in particular and must therefore consider metrics that tell us about user experience.

How does cheating affect the performance of our system, versus others, at different proportions?

In order to answer this question, we once again make use of the consistency metrics described in §5.2.5. This time, we consider the worst-case scenario. The ultimate goal of networking Virtual Environments (VEs) is synchronising state across hosts, and in the vast majority of cases, the state with the most volume and importance is avatar position. Like in our evaluation on the effect of packet loss (§5.5) we do not care about missing/extra peers in this context, for the same reasons.

Figure 5.19 shows the worst-case drift distance for different cheater probabilities across the different workloads. By "worst-case" we specifically mean that if players were moving at maximum speed in a straight line, the diameter (latency, not hops) of a computed topology causes this level of drift distance. In order to have a fairer comparison, we allow forwarding for all topologies, when in reality, the Complete and AOI topologies for example would not have optimal routing, but rather only unicast to AOI peers one hop away.

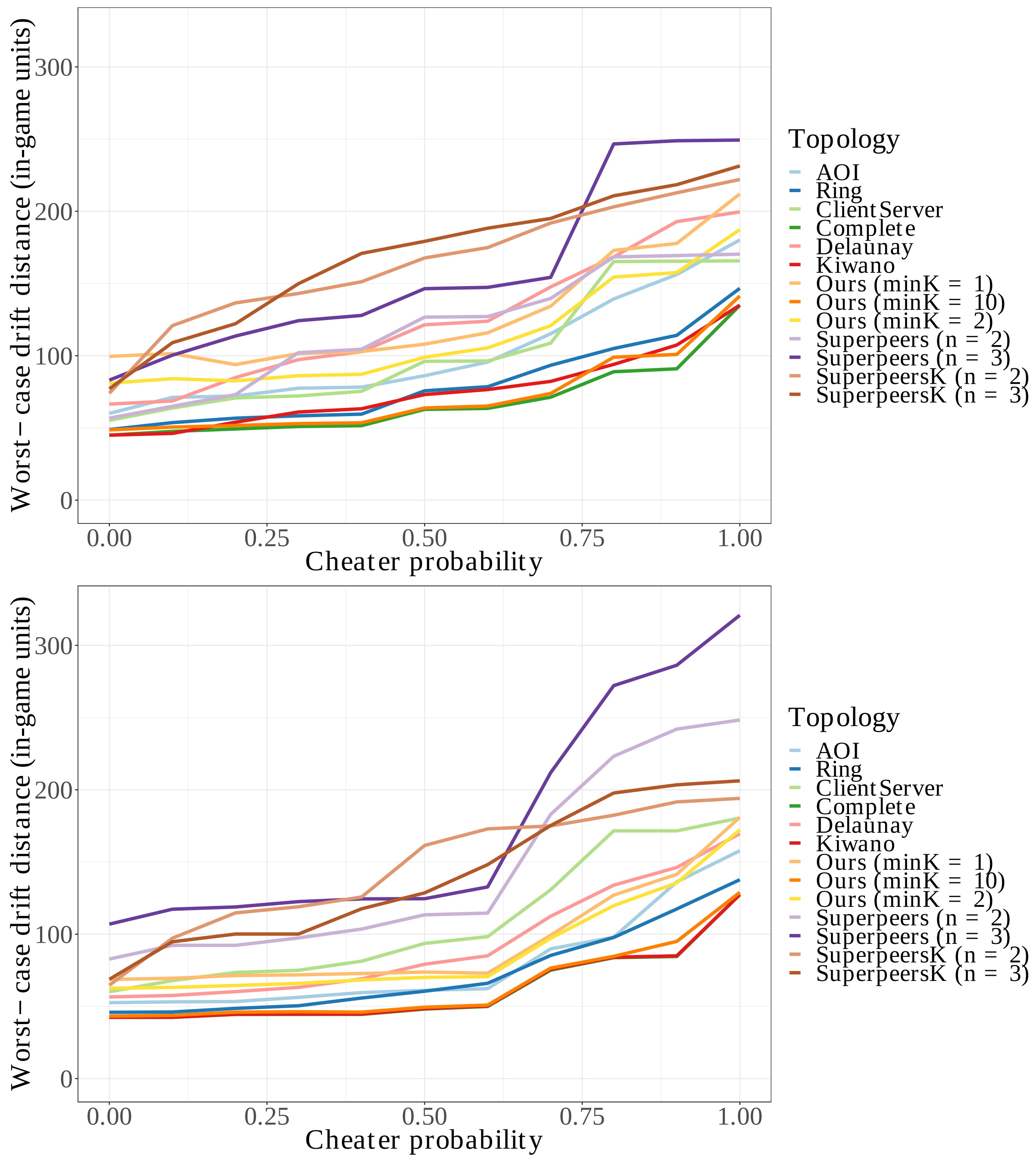


Figure 5.19: Worst case drift distances for different cheater probabilities across dynamic (top) and static (bottom) workloads

We are not so interested in how these measurements compare across topologies overall — for the most part, the patterns are as expected, where the topologies with higher connectivity are have a lower drift. Rather we are interested in how the proportion of cheaters affects these measurements. We are also interested in the spread of these measurements, which we tabulate in tables 5.4 and 5.5 across trace workloads.

Table 5.4: Table of standard deviations of worst-case drift distances for every topology (rows) and cheater probability (columns) across the dynamic workload

Topology	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1**
AOI	48.9	59.1	58.9	64.3	65.5	73.4	77.1	96.0	114.1	130.5	146.6
Ring	9.50	17.9	17.6	18.2	18.0	22.0	22.8	23.6	19.60	22.70	28.60
ClientServer	10.2	24.6	22.4	23.3	24.6	29.7	29.7	27.5	31.20	30.80	30.70
Complete	7.90	11.7	11.3	14.1	13.8	18.3	18.3	17.5	14.30	13.20	23.80
Delaunay	19.8	20.0	28.4	30.0	31.8	37.3	37.7	49.5	43.60	55.80	59.30
Kiwano	8.00	8.20	11.9	16.0	17.3	19.4	21.6	19.3	15.50	14.80	24.00
Ours (minK = 1)	40.4	38.7	29.3	29.8	30.0	30.5	34.7	42.5	50.30	51.50	58.20
Ours (minK = 2)	25.8	26.0	21.9	23.4	22.7	25.6	29.4	32.6	40.00	41.90	43.30
Ours (minK = 10)	10.3	13.0	12.5	14.7	14.8	18.7	18.8	18.0	23.40	22.30	30.70
Superpeers (n = 2)	9.20	21.0	20.9	32.2	32.3	40.9	40.8	34.2	26.00	26.40	27.50
Superpeers (n = 3)	17.3	24.7	23.2	25.5	24.0	33.6	33.4	35.3	54.50	52.30	51.80
SuperpeersK (n = 2)	23.2	48.2	57.4	58.5	60.9	61.2	61.0	68.9	68.70	70.20	69.50
SuperpeersK (n = 3)	23.6	35.9	42.1	60.8	68.7	70.3	72.3	71.3	66.50	68.40	70.80

Table 5.5: Table of standard deviations of worst-case drift distances for every topology (rows) and cheater probability (columns) across the static workload

Topology	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1**
AOI	25.8	26.6	26.7	26.9	28.1	28.7	29.3	41.5	47.6	65.7	77.50
Ring	11.6	11.6	12.2	13.2	16.1	18.8	21.9	24.3	24.5	29.5	34.70
ClientServer	22.6	36.3	37.1	39.3	38.0	45.7	53.8	86.3	70.5	70.5	67.90
Complete	9.80	9.80	10.5	10.5	10.5	11.9	12.9	19.7	19.3	19.7	29.30
Delaunay	18.8	19.2	19.4	22.4	26.7	32.4	39.6	47.9	45.8	49.7	56.50
Kiwano	9.80	9.80	10.5	10.5	10.5	12.2	13.2	20.1	19.4	19.8	29.30
Ours (minK = 1)	24.2	23.4	24.3	23.6	23.6	24.2	24.2	33.5	38.1	44.9	58.30
Ours (minK = 10)	10.2	10.6	11.2	11.3	11.2	12.6	13.4	20.6	19.6	26.1	29.90
Ours (minK = 2)	21.1	20.9	21.1	22.2	22.7	23.4	24.4	32.4	33.3	42.1	52.20
Superpeers (n = 2)	30.2	31.1	31.1	39.3	38.6	46.5	48.2	97.1	82.1	92.2	90.50
Superpeers (n = 3)	34.5	39.5	39.7	41.2	39.9	40.1	49.1	87.0	88.4	88.8	103.5
SuperpeersK (n = 2)	22.8	42.1	52.4	51.5	56.7	67.9	72.7	70.3	66.7	69.9	68.50
SuperpeersK (n = 3)	22.5	32.3	35.4	35.4	45.6	49.3	57.9	67.7	65.4	67.5	67.40

From the figures, we can see that across workloads, superpeer topologies are the ones made worse from the very first cheater, as well as the ones that are most volatile when cheaters are in the majority. This is largely due to the heterogeneity in connectivity across the network for these topologies. Especially later, when cheaters are in the majority, it is much more likely for a superpeer to become a cheater, after which all bets are off, and these topologies take a significant hit.

The ID-based superpeer selection performs significantly worse than the k-means superpeer selection here as the latter — like our system — takes account of network conditions, which does not differentiate between protocol-layer cheaters and peers with a legitimately bad internet connection. This makes it less likely that cheaters are selected as superpeers, and if they are, the diameter of the resulting topologies is still smaller that it would be for ID-based superpeer selection. The client-server architecture behaves similarly, but is much better as the server cannot be a cheater. We must also acknowledge that, in the past, superpeer architectures have incorporated reputation metrics into the superpeer selection decision, thereby lowering the likelihood that a superpeer cheats. If the Superpeers topology did the same here, then we would not see same stark increase in drift as cheaters overwhelm the network, and the line would instead be closer to SuperpeersK.

The AOI topology (with forwarding enabled) appears to perform well here, but the standard deviation tables show us that this is somewhat misleading. Under the dynamic workload, this topology becomes incredibly unpredictable as we increase the cheater probability. This is similar under the static workload, except that it is exceeded by the naive superpeer topologies, which suffer more here due to the fact that the lower churn can have the topology stuck in a suboptimal state (e.g. cheater superpeers) for longer.

The densely connected networks (Kiwano, Ring) approach our lower bound (Complete with forwarding enabled) roughly in order of connectivity as expected. It is not surprising that alternate pair-wise routes strongly mitigate the effects of cheaters.

What is surprising at first glance however, is that our topologies with $minK$ values of 1 and 2 go from performing worse than Delaunay, to better than Delaunay, *as the number of cheaters increase*. This is an important result, because it takes connectivity out of the equation (Delaunay is more connected) and if we were to consider only virtual position, then our topologies (specifically $minK = 1$) are subgraphs of Delaunay. This means that **the higher performance of our topology can be directly attributed to our neighbour selection metrics**. This effect is much more pronounced when cheaters cheat more severely, for example by delaying or dropping packets even more.

We only see the roles reversed back to what they were when we reach 100% cheaters, because our neighbour selection metrics can no longer avoid cheaters. Even in this case, the versatility of our system shines — by setting the $minK$ parameter higher (e.g. for a game where you really want to guarantee no protocol-level cheating whatsoever) our topology slowly becomes more connected and we can see its drift gets very close to our lower bound (Complete). This makes our system truly a Swiss army knife of cross-genre game networking.

5.7 Summary

In §5.1.1, we put forward a number of evaluation questions that we answered in this chapter. These evaluated our system from four angles: *scalability*, *churn sensitivity*, *loss resilience*, and *cheating mitigation*. From these, we generated a series of seven more specific questions that we address individually throughout this chapter, after outlining our setup and methodology. These were:

1. How does our system scale with number of peers, compared to others?
2. How does per-peer connection count scale for each topology as peer count increases?
3. How are upload/download rates affected as peer count increases?
4. How sensitive is our system to peer/connection churn, compared to others?
5. How resilient is our system's performance to packet loss, compared to others?
6. How influential are cheaters in our system, versus others, at different proportions?
7. How does cheating affect the performance of our system, versus others, at different proportions?

The answers to these questions are summarised through table 5.1 at the beginning of this chapter, contrasting our system with comparable alternatives. We highlight how our system is clearly the most versatile and well performing under these criteria. Since we have linked these criteria to the use case of P2P networks for browser-based NVE UD, it follows that our system is best suited to this context as it strikes a balance across workloads and performance metrics.

Chapter 6

Conclusion

At the start of this thesis, we set out to build a system that enables **Peer-to-peer (P2P) update dissemination** in a way that can be **tuned to different Networked Virtual Environment (NVE) use cases**, and that can be used under the constraints of a **web browser context**. We decomposed that goal into three parts that roughly correspond to one chapter each, after our background chapter.

1. To capture NVE, browser, and network requirements for P2P Update Dissemination (UD) systems
2. To design and implement a P2P UD system that meets these requirements
3. To evaluate our implementation and demonstrate that it does indeed meet these requirements with respect to alternative solutions

Each chapter was then dedicated to systematically proving that these goals were met. In this final chapter, we summarise how, discuss the future of this work, and conclude this thesis.

6.1 Thesis summary

In §2 we explored and reviewed high-level concepts and literature pertinent to the research areas of this PhD project. We began by looking at the history of decentralisation, the virtue of separating the control and data planes, and the browser as a platform, in order to set the scene. Then, we outlined the challenges of NVEs, and the research that is done in this area, to frame our context. We examined more detailed underlay and overlay network

research — including presenting our taxonomy of existing overlay network topologies — such that our contributions can be positioned in this bigger picture of related work. We ended this chapter with a thorough survey of evaluation metrics and a discussion of evaluation workloads, to justify our later evaluation setup and methodology.

After reviewing and surveying the literature in §2, we proceeded to meet the first goal through deep measurements and analysis of a dataset we built in §3. In this chapter, we focused on capturing NVE constraints and requirements. To do this, we built a large dataset from an existing NVE using bots and crawlers that we developed and released. We similarly made an anonymised version of this dataset publicly available. Then we captured NVE requirements through targeted measurements over this and other collected data. Along the way, we also found and disclosed high-risk vulnerabilities in the NVE we analysed, which were subsequently patched by the developers. We also ran a number of experiments and analyses to capture browser and network requirements.

We then tackled the second objective in §4. We described the design and implementation of our system, as well as the libraries and APIs we released, driven by the requirements and constraints of the previous chapter. We also implemented a range of alternative solutions, next to our own, that can be used in said library. We did this not just so that developers who use this library can have more options, but in order to be able to compare our system to others from the literature, all else being equal. We began by outlining and justifying our high-level approach, then described the system architecture and algorithms in detail.

Finally, we bet the third and final objective in §5. We developed a rich evaluation framework for P2P UD systems for NVEs, with options for evaluating under browser constraints, which we released on GitHub for other researchers to test their systems under. Then we used our framework to evaluate our system on in its own right, as well as with respect to alternate solutions. We described the evaluation of our system in depth, with respect to different workloads, parameters, and existing solutions, in order to meet specific research goals.

This lead us to verify that our system did indeed solve the problems we sought out to solve, to an extent that met our constraints/requirements, and also surpassed alternate solutions. Further, we have shown that our system is versatile enough that it can be configured to match a wide range of use cases, rather than be overly specialised for a single one, making it generalisable without the concessions of trade-offs. We truly believe that the contributions in technology as well as research and development resources that we have made in the course of this PhD will have a profound and long-lived impact on the field.

6.2 Future work

At the time of writing this, the COVID-19 pandemic is sweeping the globe and the future is uncertain. We are seeing a spike in reliance on video conferencing, collaborative editing, streaming, and online games. We have seen platforms like Zoom suffer outages and reliability issues amid this deluge of demand. Ironically, Zoom has previously dismissed P2P as not scalable [143]. Similarly, Microsoft centralised the Skype protocol in 2012, which used a superpeer architecture, citing performance, scalability, and availability benefits [50]. In 2013, it turned out that the real motivation reason could have been the NSA's PRISM surveillance programme [41, 52].

All this is to say that real-time P2P, and therefore our systems, have increasingly viable horizontal applications. Our prototype web apps that we described in §4.8 are an example of this. Indeed, many of these use cases can be browser-based, as for example video transcoding has become practical in the browser.

In the gaming space, online games have seen a large COVID influx now too, Manyland included. Steam sales are through the roof as people who have never played video games before are exploring the hobby while self-isolating. This makes our solution even more desirable to tackle scalability, performance, and fault-tolerance. UD is not the only requirement in NVEs that benefits from distribution. As we touched on earlier, persistent storage (both static assets and player-generated content) is another such high-potential area. Existing distributed storage solutions are general-purpose, for example IPFS [5], and could benefit from context/use case awareness and application-layer metrics the same way that our UD system has. Different vertical approaches like this would enable us to offer complete game networking solutions that decentralises/distributes as much as can be (and makes sense to) distribute.

The dataset we built and present in §3 also opens a number of different research avenue's. Besides looking at NVE mobility through a behavioural / game design lens, this data can be used for social networks analysis, natural language processing (through chat logs), and developing game AI. Solving network problems in only one of many problems that can be tackled and solved through further deep analysis of our dataset.

Furthermore, we hope that our browser library and signalling server API sees mass adoption, for example through the rising `.io` game trend. These often "fake" multiplayer using bots because real multiplayer is expensive and complex to develop [103]. This would enable us to tackle a range of even more difficult research questions, for example finding out which and what proportion of peers are unable to form P2P connections and fall back to using TURN servers due to NAT traversal issues. This, and other usage data would

allow us to build even better systems, as well as flood the research community with even more large, rich analytics and datasets. We hope that this thesis is just the beginning in a greater endeavour to level the developer playing field and catalyse the creation of better applications.

Bibliography

[1] Nils Aschenbruck et al. "BonnMotion: a mobility scenario generation and analysis tool". In: *Proceedings of the 3rd international ICST conference on simulation tools and techniques*. ICST (Institute for Computer Sciences, Social-Informatics and ... 2010, p. 51.

[2] Franz Aurenhammer. "Voronoi diagrams—a survey of a fundamental geometric data structure". In: *ACM Computing Surveys (CSUR)* 23.3 (1991), pp. 345–405.

[3] Helge Backhaus, Stephan Krause, et al. "QuON: a quad-tree-based overlay protocol for distributed virtual worlds". In: *International Journal of Advanced Media and Communication* 4.2 (2010), p. 126.

[4] Ricky A Bangun, Eryk Dutkiewicz, and Gary J Anido. "An analysis of multi-player network games traffic". In: *1999 IEEE Third Workshop on Multimedia Signal Processing (Cat. No. 99TH8451)*. IEEE. 1999, pp. 3–8.

[5] Juan Benet. "Ipfs-content addressed, versioned, p2p file system". In: *arXiv preprint arXiv:1407.3561* (2014).

[6] Steve Benford and Lennart Fahlén. "A spatial model of interaction in large virtual environments". In: *Proceedings of the Third European Conference on Computer-Supported Cooperative Work 13–17 September 1993, Milan, Italy ECSCW'93*. Springer. 1993, pp. 109–124.

[7] Carlos Eduardo Bezerra, Fábio R Cecin, and Cláudio FR Geyer. "A3: A novel interest management algorithm for distributed simulations of mmogs". In: *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*. IEEE Computer Society. 2008, pp. 35–42.

[8] Carlos Eduardo Benevides Bezerra, João Luiz Dihl Comba, and Cláudio Fernando Resin Geyer. "A fine granularity load balancing technique for MMOG servers using a kd-tree to partition the space". In: *2009 VIII Brazilian Symposium on Games and Digital Entertainment*. IEEE. 2009, pp. 17–26.

[9] Ashwin Bharambe et al. "Donnybrook: enabling large-scale, high-speed, peer-to-peer games". In: *ACM SIGCOMM Computer Communication Review* 38.4 (2008), pp. 389–400.

[10] Ashwin R Bharambe, Jeffrey Pang, and Srinivasan Seshan. "Colyseus: A Distributed Architecture for Online Multiplayer Games." In: *NSDI*. Vol. 6. 2006, pp. 12–12.

[11] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[12] Zack Bloom. *Serverless Performance: Cloudflare Workers, Lambda and Lambda@Edge.* 2018. URL: `https://blog.cloudflare.com/serverless-performance-comparison-workers-lambda/` (visited on 07/21/2019).

[13] Jean-Sébastien Boulanger, J"org Kienzle, and Clark Verbrugge. "Comparing interest management algorithms for massively multiplayer games". In: *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games.* ACM. 2006, p. 6.

[14] Eliya Buyukkaya and Maha Abdallah. "Efficient triangulation for p2p networked virtual environments". In: *Multimedia Tools and Applications* 45.1-3 (2009), pp. 291–312.

[15] Eliya Buyukkaya, Maha Abdallah, and Romain Cavagna. "VoroGame: a hybrid P2P architecture for massively multiplayer games". In: *2009 6th IEEE Consumer Communications and Networking Conference.* Ieee. 2009, pp. 1–5.

[16] Jean de Campredon et al. "Hybridearth: Social mixed reality at planet scale". In: *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC).* IEEE. 2014, pp. 1138–1139.

[17] Emanuele Carlini, Massimo Coppola, and Laura Ricci. "Integration of P2P and clouds to support massively multiuser virtual environments". In: *2010 9th Annual Workshop on Network and Systems Support for Games.* IEEE. 2010, pp. 1–6.

[18] Miguel Castro et al. "SCRIBE: A large-scale and decentralized application-level multicast infrastructure". In: *IEEE Journal on Selected Areas in communications* 20.8 (2002), pp. 1489–1499.

[19] Miguel Castro et al. "Secure routing for structured peer-to-peer overlay networks". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 299–314.

[20] Romain Cavagna, Christian Bouville, and Jerome Royan. "P2P network for very large virtual environment". In: *Proceedings of the ACM symposium on Virtual reality software and technology.* ACM. 2006, pp. 269–276.

[21] Kuan-Ta Chen et al. "Game traffic analysis: An MMORPG perspective". In: *Proceedings of the international workshop on Network and operating systems support for digital audio and video.* ACM. 2005, pp. 19–24.

[22] Yang Chen et al. "Pharos: accurate and decentralised network coordinate system". In: *IET communications* 3.4 (2009), pp. 539–548.

[23] Yang Chen et al. "Phoenix: Towards an accurate, practical and decentralized network coordinate system". In: *International Conference on Research in Networking.* Springer. 2009, pp. 313–325.

[24] Alice Cheng and Eric Friedman. "Sybilproof reputation mechanisms". In: *Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems.* ACM. 2005, pp. 128–132.

[25] T Matthew Ciolek and Adam Kendon. "Environment and the spatial arrangement of conversational encounters". In: *Sociological Inquiry* 50.3-4 (1980), pp. 237–271.

[26] Mark Claypool and Kajal Claypool. "On latency and player actions in online games". In: (2006).

[27] Trevor F Cox and Michael AA Cox. *Multidimensional scaling*. Chapman and hall/CRC, 2000.

[28] Frank Dabek et al. "Vivaldi: A decentralized network coordinate system". In: *ACM SIGCOMM Computer Communication Review*. Vol. 34. 4. ACM. 2004, pp. 15–26.

[29] Frank Dabek et al. "Wide-area cooperative storage with CFS". In: *ACM SIGOPS Operating Systems Review*. Vol. 35. 5. ACM. 2001, pp. 202–215.

[30] George Danezis et al. "Sybil-resistant DHT routing". In: *European Symposium On Research In Computer Security*. Springer. 2005, pp. 305–318.

[31] Alan Demers et al. "Epidemic algorithms for replicated database maintenance". In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 1987, pp. 1–12.

[32] Raluca Diaconu and Joaquín Keller. "Kiwano: A scalable distributed infrastructure for virtual worlds". In: *2013 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2013, pp. 664–667.

[33] Christophe Diot and Laurent Gautier. "A distributed architecture for multiplayer interactive applications on the Internet". In: *IEEE network* 13.4 (1999), pp. 6–15.

[34] Benoit Donnet, Bamba Gueye, and Mohamed Ali Kaafar. "A survey on network coordinates systems, design, and security". In: *IEEE Communications Surveys & Tutorials* 12.4 (2010), pp. 488–503.

[35] John R Douceur. "The sybil attack". In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 251–260.

[36] Wu-chang Feng et al. "A traffic characterization of popular on-line games". In: *IEEE/ACM Transactions on Networking (TON)* 13.3 (2005), pp. 488–500.

[37] Linton C Freeman. "A set of measures of centrality based on betweenness". In: *Sociometry* (1977), pp. 35–41.

[38] Davide Frey et al. "Solipsis: A decentralized architecture for virtual environments". In: *1st International Workshop on Massively Multiuser Virtual Environments*. 2008.

[39] Mark Frohnmayer and Tim Gift. "The TRIBES engine networking model". In: *Proceedings of the Game Developers Conference*. 2000.

[40] Thomas MJ Fruchterman and Edward M Reingold. "Graph drawing by force-directed placement". In: *Software: Practice and experience* 21.11 (1991), pp. 1129–1164.

[41] Ryan Gallagher. *Newly Revealed PRISM Snooping Makes Verizon Surveillance Look Like Kids' Stuff*. 2013. URL: `https://slate.com/technology/2013/06/nsa-prism-surveillance-private-data-from-google-microsoft-skype-apple-yahoo-snooped-by-government.html` (visited on 03/20/2020).

[42] Game Developers Conference (GDC). *State of the Game Industry 2017*. Tech. rep. 2017.

[43] Game Developers Conference (GDC). *State of the Game Industry 2018*. Tech. rep. 2018.

[44] Paul Gardner, Margo Seltzer, et al. "Network Coordinates in the Wild". In: *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*. 2007.

[45] Chris GauthierDickey, Virginia Lo, and Daniel Zappala. "Using n-trees for scalable event ordering in peer-to-peer games". In: *Proceedings of the international workshop on Network and operating systems support for digital audio and video*. ACM. 2005, pp. 87–92.

[46] Chris GauthierDickey, Daniel Zappala, and Virginia Lo. "Event Ordering and Congestion Control for Distributed Multiplayer Games". In: *article), May* 14 (2005), p. 10.

[47] Chris GauthierDickey et al. "Low latency and cheat-proof event ordering for peer-to-peer games". In: *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*. ACM. 2004, pp. 134–139.

[48] Luca Genovali and Laura Ricci. "Voronoi models for distributed virtual environments". In: *Proceedings of the 2008 ACM CoNEXT Conference*. ACM. 2008, p. 43.

[49] John S Gilmore and Herman A Engelbrecht. "A survey of state persistency in peer-to-peer massively multiplayer online games". In: *IEEE Transactions on Parallel and Distributed Systems* 23.5 (2011), pp. 818–834.

[50] Dan Goodin. *Skype replaces P2P supernodes with Linux boxes hosted by Microsoft*. 2012. URL: https://arstechnica.com/information-technology/2012/05/skype-replaces-p2p-supernodes-with-linux-boxes-hosted-by-microsoft/ (visited on 03/20/2020).

[51] Chris Greenhalgh and Steve Benford. "MASSIVE: a distributed virtual reality system incorporating spatial trading". In: *Proceedings of 15th International Conference on Distributed Computing Systems*. IEEE. 1995, pp. 27–34.

[52] Glenn Greenwald. *NSA Prism program taps in to user data of Apple, Google and others*. 2013. URL: https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data (visited on 03/20/2020).

[53] Ian J Grimstead, Nick J Avis, and David W Walker. "RAVE: Resource-Aware Visualization Environment". In: (2006).

[54] Ian J Grimstead, Nick J Avis, and David W Walker. "RAVE: the resource-aware visualization environment". In: *Concurrency and Computation: Practice and Experience* 21.4 (2009), pp. 415–448.

[55] Krishna P Gummadi, Stefan Saroiu, and Steven D Gribble. "King: Estimating latency between arbitrary internet end hosts". In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM. 2002, pp. 5–18.

[56] Hakim Hacid et al. "Enhancing navigation in virtual worlds through social networks analysis". In: *International Symposium on Methodologies for Intelligent Systems*. Springer. 2011, pp. 146–152.

[57] Thorsten Hampel, Thomas Bopp, and Robert Hinn. “A peer-to-peer architecture for massive multiplayer online games”. In: *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. ACM. 2006, p. 48.

[58] Tom Hardin. *2018 Digital Trend: Microservices*. 2018. URL: https://blog.g2crowd.com/blog/trends/digital-platforms/2018-dp/microservices/ (visited on 05/14/2019).

[59] Scott Hendrickson et al. “Serverless computation with openlambda”. In: *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. 2016.

[60] Shun-Yun Hu, Shao-Chen Chang, and Jehn-Ruey Jiang. “Voronoi state management for peer-to-peer massively multiplayer online games”. In: *2008 5th IEEE Consumer Communications and Networking Conference*. IEEE. 2008, pp. 1134–1138.

[61] Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. “VON: a scalable peer-to-peer network for virtual environments”. In: *IEEE Network* 20.4 (2006), pp. 22–31.

[62] Shun-Yun Hu and Guan-Ming Liao. “Scalable peer-to-peer networked virtual environment”. In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. ACM. 2004, pp. 129–133.

[63] Shun-Yun Hu et al. “A spatial publish subscribe overlay for massively multiuser virtual environments”. In: *2010 International Conference on Electronics and Information Engineering*. Vol. 2. IEEE. 2010, pp. V2–314.

[64] Guan-Yu Huang, Shun-Yun Hu, and Jehn-Ruey Jiang. “Scalable reputation management with trustworthy user selection for P2P MMOGs”. In: *International Journal of Advanced Media and Communication* 2.4 (2008), pp. 380–401.

[65] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. “Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games”. In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. ACM. 2004, pp. 116–120.

[66] Laura Itzel et al. “Specifying consistency requirements for massively multi-user virtual environments”. In: *2011 IEEE International Workshop on Haptic Audio Visual Environments and Games*. IEEE. 2011, pp. 1–2.

[67] Laura Itzel et al. “The quest for meaningful mobility in massively multi-user virtual environments”. In: *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*. IEEE Press. 2011, p. 15.

[68] Josh James. *Data Never Sleeps 2.0*. 2014. URL: https://www.domo.com/blog/2014/04/data-never-sleeps-2-0/ (visited on 03/09/2016).

[69] Jared Jardine and Daniel Zappala. “A hybrid architecture for massively multiplayer online games”. In: *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*. ACM. 2008, pp. 60–65.

[70] Jehn-Ruey Jiang, Yu-Li Huang, and Shun-Yun Hu. “Scalable AOI-cast for peer-to-peer networked virtual environments”. In: *2008 The 28th International Conference on Distributed Computing Systems Workshops*. IEEE. 2008, pp. 447–452.

[71] David B Johnson and David A Maltz. “Dynamic source routing in ad hoc wireless networks”. In: *Mobile computing*. Springer, 1996, pp. 153–181.

[72] State of JS. *The State of JavaScript 2018: Front-end Frameworks - Overview*. 2019. URL: https://2018.stateofjs.com/front-end-frameworks/overview/ (visited on 07/20/2019).

[73] Jonathan Jungck. *Mobile Gaming: The Fake Multiplayer Epidemic*. 2018. URL: https://www.linkedin.com/pulse/mobile-gaming-fake-multiplayer-epidemic-jonathan-jungck/ (visited on 10/18/2020).

[74] Kleomenis Katevas et al. “Detecting group formations using iBeacon technology”. In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM. 2016, pp. 742–752.

[75] Leo Katz. “A new status index derived from sociometric analysis”. In: *Psychometrika* 18.1 (1953), pp. 39–43.

[76] Hanna Kavalionak et al. “Integrating peer-to-peer and cloud computing for massively multiuser online games”. In: *Peer-to-Peer Networking and Applications* 8.2 (2015), pp. 301–319.

[77] Yoshihiro Kawahara, Tomonori Aoyama, and Hiroyuki Morikawa. “A peer-to-peer message exchange scheme for large-scale networked virtual environments”. In: *Telecommunication Systems* 25.3-4 (2004), pp. 353–370.

[78] Joaquín Keller and Raluca Diaconu. “OneSim: Scaling Second Life with Kiwano”. In: *MMVE*. 2014, pp. 1–2.

[79] Joaquín Keller and Mathilde Laurent. “Divereal: a social virtual world without rooms”. In: *Proceedings of the 15th Annual Workshop on Network and Systems Support for Games*. IEEE Press. 2017, pp. 46–48.

[80] Joaquín Keller and Gwendal Simon. “Toward a peer-to-peer shared virtual reality”. In: *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*. IEEE. 2002, pp. 695–700.

[81] Adam Kendon. “13 The negotiation of context in face-to-face interaction”. In: *Rethinking context: Language as an interactive phenomenon* 11 (1992), p. 323.

[82] Adam Kendon. *Conducting interaction: Patterns of behavior in focused encounters*. Vol. 7. CUP Archive, 1990.

[83] Alan Kenny, Séamus Mcloone, and Tomás Ward. “Controlling entity state updates to maintain remote consistency within a distributed interactive application”. In: *ACM Transactions on Internet Technology (TOIT)* 9.4 (2009), p. 15.

[84] Kyoung-chul Kim, Ikjun Yeom, and Joonwon Lee. “Hyms: A hybrid mmog server architecture”. In: *IEICE transactions on information and systems* 87.12 (2004), pp. 2706–2713.

[85] Bjorn Knutsson et al. “Peer-to-peer support for massively multiplayer games”. In: *IEEE INFOCOM 2004*. Vol. 1. IEEE. 2004.

[86] kovarex. *Friday Facts #147 - Multiplayer rewrite*. 2016. URL: https://www.factorio.com/blog/post/fff-147 (visited on 10/10/2019).

[87] Santosh Kulkarni. "Badumna network suite: A decentralized network engine for massively multiplayer online applications". In: *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE. 2009, pp. 178–183.

[88] Chi-Anh La and Pietro Michiardi. "Characterizing user mobility in second life". In: *Proceedings of the first workshop on Online social networks*. ACM. 2008, pp. 79–84.

[89] Renaud Lambiotte, J-C Delvenne, and Mauricio Barahona. "Laplacian dynamics and multiscale modular structure in networks". In: *arXiv preprint arXiv:0812.1770* (2008).

[90] Youngki Lee et al. "Measurement and estimation of network QoS among peer Xbox 360 game players". In: *International Conference on Passive and Active Network Measurement*. Springer. 2008, pp. 41–50.

[91] Sergey Legtchenko, Sébastien Monnet, and Ga¨el Thomas. "Blue Banana: resilience to avatar mobility in distributed MMOGs". In: *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2010, pp. 171–180.

[92] Chris Lesniewski-Laas. "A Sybil-proof one-hop DHT". In: *Proceedings of the 1st workshop on Social network systems*. ACM. 2008, pp. 19–24.

[93] Huiguang Liang et al. "Avatar mobility in user-created networked virtual worlds: measurements, analysis, and implications". In: *Multimedia Tools and Applications* 45.1-3 (2009), pp. 163–190.

[94] Harsha V Madhyastha et al. "iPlane: An information plane for distributed services". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 367–380.

[95] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. "DHT routing using social links". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2004, pp. 100–111.

[96] Shawn Martin et al. "OpenOrd: an open-source toolbox for large graph layout". In: *Visualization and Data Analysis 2011*. Vol. 7868. International Society for Optics and Photonics. 2011, p. 786806.

[97] Matheus28. *WebRTC: the future of web games – Hacker News*. 2016. URL: `https://news.ycombinator.com/item?id=13264952` (visited on 10/10/2019).

[98] Nobutaka Matsumoto et al. "A scalable and low delay communication scheme for networked virtual environments". In: *IEEE Global Telecommunications Conference Workshops, 2004. GlobeCom Workshops 2004*. IEEE. 2004, pp. 529–535.

[99] Petar Maymounkov and David Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[100] Peyton Maynard-Koran. *Fixing the Internet for Real Time Applications: Part I*. 2015. URL: `https://technology.riotgames.com/news/fixing-internet-real-time-applications-part-i` (visited on 07/18/2019).

[101] Peyton Maynard-Koran. *Fixing the Internet for Real Time Applications: Part II.* 2016. URL: https://technology.riotgames.com/news/fixing-internet-real-time-applications-part-ii (visited on 07/18/2019).

[102] Peyton Maynard-Koran. *Fixing the Internet for Real Time Applications: Part III.* 2016. URL: https://technology.riotgames.com/news/fixing-internet-real-time-applications-part-iii (visited on 07/18/2019).

[103] Mizizizi. *.io Games and The Rise of Fake Multiplayer*. 2020. URL: https://www.youtube.com/watch?v=YCqnD40Q5T8 (visited on 03/19/2020).

[104] Alberto Montresor and Márk Jelasity. "PeerSim: A scalable P2P simulator". In: *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE. 2009, pp. 99–100.

[105] Katherine L Morse et al. *Interest management in large-scale distributed simulations*. Information and Computer Science, University of California, Irvine, 1996.

[106] TS Ng and Hui Zhang. "Towards global network positioning". In: *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. Citeseer. 2001, pp. 25–29.

[107] Jauvane C De Oliveira and Nicolas D Georganas. "VELVET: An adaptive hybrid architecture for very large virtual environments". In: *Presence: Teleoperators & Virtual Environments* 12.6 (2003), pp. 555–580.

[108] Jeremy Peel. *Gabe takes to Reddit to clear up Valve Anti-Cheat rumours*. 2011. URL: https://www.pcgamesn.com/counterstrike/gabe-takes-reddit-clear-valve-anti-cheat-rumours-do-we-send-your-browsing-history-valve-no (visited on 11/02/2019).

[109] Bogdan Popescu. "Safe and private data sharing with turtle: friends team-up and beat the system (transcript of discussion)". In: *International Workshop on Security Protocols*. Springer. 2004, pp. 221–230.

[110] Ananth Rao et al. "Load balancing in structured p2p systems". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 68–79.

[111] Sylvia Ratnasamy et al. *A scalable content-addressable network*. Vol. 31. 4. ACM, 2001.

[112] Laura Ricci et al. "Aoi-cast by compass routing in delaunay based dve overlays". In: *2011 International Conference on High Performance Computing & Simulation*. IEEE. 2011, pp. 135–142.

[113] Andrea W Richa, M Mitzenmacher, and R Sitaraman. "The power of two random choices: A survey of techniques and results". In: *Combinatorial Optimization* 9 (2001), pp. 255–304.

[114] Bj"orn Richerzhagen et al. "Bypassing the cloud: Peer-assisted event dissemination for augmented reality games". In: *14-th IEEE International Conference on Peer-to-Peer Computing*. IEEE. 2014, pp. 1–10.

[115] Sean Rooney, Daniel Bauer, and Rudy Deydier. "A federated peer-to-peer network game architecture". In: *IEEE Communications Magazine* 42.5 (2004), pp. 114–122.

[116] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350.

[117] Gregor Schiele et al. "Requirements of peer-to-peer-based massively multiplayer online gaming". In: *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*. IEEE. 2007, pp. 773–782.

[118] Arne Schmieg et al. "pSense-Maintaining a dynamic localized peer-to-peer structure for position based multicast in games". In: *2008 Eighth International Conference on Peer-to-Peer Computing*. IEEE. 2008, pp. 247–256.

[119] Marc Shapiro et al. "Conflict-free replicated data types". In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[120] Atul Singh et al. "Eclipse attacks on overlay networks: Threats and defenses". In: *In IEEE INFOCOM*. Citeseer. 2006.

[121] StatCounter Global Stats. *Browser market share*. 2019. URL: `https://netmarketshare.com/browser-market-share.aspx` (visited on 07/20/2019).

[122] StatCounter Global Stats. *Browser Market Share Worldwide*. 2019. URL: `http://gs.statcounter.com/browser-market-share#monthly-201811-201811-bar` (visited on 07/20/2019).

[123] StatCounter Global Stats. *Desktop vs Mobile vs Tablet Market Share Worldwide*. 2019. URL: `http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet` (visited on 07/20/2019).

[124] Ion Stoica et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Transactions on Networking (TON)* 11.1 (2003), pp. 17–32.

[125] Ivan Stojmenovic. "Position-based routing in ad hoc networks". In: *IEEE communications magazine* 40.7 (2002), pp. 128–134.

[126] Swee Ann Tan, William Lau, and Allan Loh. "Networked game mobility model for first-person-shooter games". In: *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. ACM. 2005, pp. 1–9.

[127] Vinod Tandon and Jake Devore. *Detecting lag switch cheating in game*. US Patent App. 15/585,111. Oct. 2017.

[128] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[129] Tonio Triebel et al. "Peer-to-peer infrastructures for games". In: *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM. 2008, pp. 123–124.

[130] Mathieu Valero, Raluca Diaconu, and Joaquín Keller. "Manycraft: Massively distributed minecraft". In: *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*. IEEE. 2013, pp. 1–3.

[131] Matteo Varvello, Ernst Biersack, and Christophe Diot. "A networked virtual environment over KAD". In: *Proceedings of the 2007 ACM CoNEXT conference*. ACM. 2007, p. 66.

[132] Matteo Varvello et al. "Distributed avatar management for second life". In: *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*. IEEE Press. 2009, p. 5.

[133] Matteo Varvello et al. "Is there life in Second Life?" In: *Proceedings of the 2008 ACM CoNEXT Conference*. ACM. 2008, p. 1.

[134] Voodoo. *Bots - Hole.io FAQ*. 2020. URL: `https://hole-io.com/faq/bots.php` (visited on 10/18/2020).

[135] W3Techs. *Usage of server-side programming languages for websites*. 2019. URL: `https://w3techs.com/technologies/overview/programming_language/all` (visited on 06/19/2019).

[136] Yichuan Wang et al. "Network traces of virtual worlds: Measurements and applications". In: *Proceedings of the second annual ACM conference on Multimedia systems*. ACM. 2011, pp. 105–110.

[137] Amir Yahyavi and Bettina Kemme. "Peer-to-peer architectures for massively multiplayer online games: A survey". In: *ACM Computing Surveys (CSUR)* 46.1 (2013), p. 9.

[138] B Beverly Yang and Hector Garcia-Molina. "Designing a super-peer network". In: *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE. 2003, pp. 49–60.

[139] Anthony Peiqun Yu and Son T Vuong. "MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games". In: *Proceedings of the international workshop on Network and operating systems support for digital audio and video*. Acm. 2005, pp. 99–104.

[140] Ben Y Zhao et al. "Tapestry: A resilient global-scale overlay for service deployment". In: *IEEE Journal on selected areas in communications* 22.1 (2004), pp. 41–53.

[141] Jin Zhou et al. "A low-latency peer-to-peer approach for massively multiplayer games". In: *International Workshop on Agents and P2P Computing*. Springer. 2005, pp. 120–131.

[142] Xinyu Zhuang et al. "Player dynamics in massively multiplayer online games". In: (2007).

[143] Zoom. *Cloud-Based and Peer-to-Peer Meetings*. 2014. URL: `https://blog.zoom.us/wordpress/2014/10/09/cloud-based-and-peer-peer-meetings/` (visited on 03/20/2020).

Glossary

AFK Away From Keyboard. 50

ALM Application-Layer Multicast. 24, 33, 59, 109, 115, 116, 163, 164, 172

AOI Area-of-Interest. ix–xi, 17, 20, 21, 24, 26, 29–40, 45, 46, 49, 51, 55, 56, 58–60, 64, 72, 77, 78, 89, 108–110, 113, 114, 116, 118, 124, 125, 128, 131, 135–137, 145, 150–159, 161, 163–167, 173, 176, 178, 179

AR Augmented Reality. 24, 34, 133

AWS Amazon Web Services. 12

CDN Content Delivery Network. 13

CRDT Conflict-free Replicated Data Type. 22

CRUD Create, Read, Update, and Delete. 12

CSP Content Security Policy. 11

CSR Client-Side Rendering. 9, 11, 15

DHT Distributed Hash Table. 3, 9, 10, 20, 22, 27, 29, 30, 32, 34, 112, 115, 135

EMST Euclidean Minimum Spanning Tree. 126

FPS First-Person Shooter. 4, 15, 17, 19, 22, 40, 47, 52, 66, 133, 151

ICE Interactive Connectivity Establishment. 131

IM Interest Management. 17, 18, 27, 31, 46, 47, 55, 119, 156

IoT Internet of Things. 1, 11, 15

IRC Internet Relay Chat. 24

LOD Level of Detail. 26

LOESS locally estimated scatterplot smoothing. 55, 58, 81, 160, 170

MANET Mobile Ad hoc Network. 22, 24

MMOG Massively Multiplayer Online Game. 4, 15, 17, 22, 23, 36, 43, 45, 47, 48, 52, 56, 68, 69, 133, 140, 156

MMORPG Massively Multiplayer Online Role-Playing Game. 22

MOBA Multiplayer Online Battle Arena. 133, 140

MST Minimum Spanning Tree. 126, 163

NAT Network Address Translation. 106

NCS Network Coordinate System. 25, 32, 34, 119, 123, 137

NLOD Network Level of Detail. 26, 140

NVE Networked Virtual Environment. ix, 1–8, 10, 16–18, 22, 39, 44–46, 48, 52, 60, 64, 72, 83, 88, 108, 109, 111, 112, 130, 133, 135, 141, 142, 151, 156, 159, 163, 171, 173, 179, 183–186

OM Object Management. 21, 27, 30, 31, 33, 135

P2P Peer-to-peer. ix, xi, 2–6, 9, 10, 13, 14, 16–24, 26–29, 31, 32, 34–36, 38–41, 44–47, 52, 54, 64, 68, 69, 72, 83, 85, 87–89, 92, 100, 101, 103, 104, 107–109, 111–114, 125–127, 130, 133, 135, 136, 138, 142, 150, 152, 156, 159, 161, 163, 171, 173, 183–186

PoP Point of Presence. xi, 147, 148

PvP player versus player. 52

PWA Progressive Web App. 9, 12

QoS Quality of Service. 40, 41

REST Representational State Transfer. 11, 12, 15, 89

RPG Role-Playing Game. 17

RTP Real-time Transport Protocol. 104

RTS Real-Time Strategy. 17, 19, 133

RTT Round-Trip Time. 42, 106

RWP Random Waypoint Model. 22, 151, 153, 161

SAN Storage Area Network. 10

SDN Software-Defined Networking. 1, 10, 111

SDP Session Description Protocol. 131

SPA Single-Page Application. 9, 11, 12

UD Update Dissemination. 4, 21, 27, 29–31, 33, 35, 36, 41, 44, 45, 88, 114–116, 135, 136, 142, 159, 171, 183–186

UX User Experience. 40

VE Virtual Environment. 14, 17, 18, 20–23, 27, 29–31, 34, 40, 41, 47, 66, 77, 100, 103, 119, 122, 179

VPS Virtual Private Server. 12

VR Virtual Reality. 1, 2, 15

XSS Cross-Site Scripting. 5, 12, 83