Implementation of a Peer-to-Peer Spatial Publish/Subscribe System for Networked Virtual Environments Using BubbleStorm

Master-Thesis from Marcel Lucas
May 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik

DVS

Implementation of a Peer-to-Peer Spatial Publish/Subscribe System for Networked Virtual Environments Using BubbleStorm

Accepted Master-Thesis from Marcel Lucas

1. Assessor: Prof. Alejandro P. Buchmann, Ph. D.
2. Assessor:

Date of Submission:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 17. Mai 2011

(Marcel Lucas)

Abstract

The popularity of Networked Virtual Environments (NVE) increased remarkably in recent years where a plethora of players participate concurrently. Traditional approaches applying the Client/Server pattern swiftly reach their limit in terms of scalability and maintainability. Hence, different approaches are based on Peer-to-Peer (P2P) technology. In particular, fast-paced NVEs stress these systems and most likely cause high overlay maintenance overhead.

This Master's Thesis presents a novel approach to spatial Publish/Subscribe (Pub/Sub) in P2P NVEs. The unstructured P2P system BubbleStorm offers flexible and scalable means and is therefore used as a fundamental base for the developed Pub/Sub system. As the evaluation of the subscription is performed at the peers and not on the overlay, the application is free to use every query language in any complexity. Subscriptions are disseminated into the overlay by BubbleStorm and updated periodically due to the volatile nature of spatial data. Publications are, for the sake of near-real-time delivery, sent by applying direct connectivity between end points. To further minimize latency, CUSP is used as the transport protocol for all network communication. Consequently, area subscriptions with point publications are achieved. The developed overlay is evaluated with the massively multiplayer online game prototype Planet $\pi 4$ which features an open world scenario with support for plenty of concurrent players. The novel Pub/Sub approach is compared against a traditional Client/Server approach as well as the P2P system pSense to determine performance.

Zusammenfassung

Die Popularität von virtuellen verteilten Welten (Networked Virtual Environments; NVE) mit einer Vielzahl gleichzeitiger Spieler stieg in den letzten Jahren deutlich an. Traditionelle Ansätze wie Client/Server (C/S) stoßen dabei schnell an die Grenze des Machbaren, vor allem in Bezug auf Skalierbarkeit und Wartbarkeit. Aus diesem Grund existieren viele Ansätze, die auf Peer-to-Peer (P2P) Technologie setzen. Besonders in NVE mit schneller Spielmechanik wird das zugrundeliegende Netzwerk stark beansprucht und verursacht einen großen Verwaltungsaufwand.

Diese Master-Thesis präsentiert einen neuartigen Ansatz für räumliches Veröffentlichen und Abonnieren (Publish/Subscribe; Pub/Sub) auf Basis von P2P Technologie. Dazu wird das unstrukturierte BubbleStorm-System eingesetzt welches gute Skalierbarkeit und Flexibilität aufweist. Abonnements können hierbei eine beliebige Komplexität aufweisen und werden durch BubbleStorm periodisch im Netzwerk verteilt. Die Veröffentlichung von Daten geschieht durch direktes Senden an den Kommunikationspartner um eine möglichst geringe Latenz zu erreichen. Die Latenz kann weiterhin durch den Einsatz des CUSP Protokolls, welches für den gesamten Netzwerkverkehr eingesetzt wird, minimiert werden. Insgesamt wird auf diese Weise ein System erreicht, welches räumliche Abonnements mit punktuellen Veröffentlichungen bietet. Das entwickelte System wird mit Hilfe des Massen-Mehrspieler-Online-Gemeinschaftsspiels (massively multiplayer online game; MMOG) Planet $\pi 4$, das eine offene virtuelle Welt mit Unterstützung für eine Vielzahl gleichzeitiger Spieler bietet, ausgewertet und auf Tauglichkeit eines Pub/Sub-Ansatzes überprüft. Das entwickelte Pub/Sub-System wird außerdem mit dem traditionellen C/S-Ansatz sowie dem P2P-System pSense verglichen.

Contents

Glossary

Notation	Description
ALM	Application Layer Multicast; A distributed system that can efficiently handle broadcasts to a high number of peers.
AOI	Area of Interest; The area surrounding a player including objects or other players that are of interest.
C/S	Client-Server; A distributed networking model with a central instance and directly connected clients.
DHT	Distributed Hash Table; A distributed P2P-system that offers hash table-like functionality.
FPS	First-Person Shooter; A video game that focusses on a first-person perspective gameplay.
MMOG	Massively Multiplayer Online Game; A multiplayer game with a high number of concurrent players.
Node	Node — or peer — defines an entity which takes part in an P2P networking overlay. Preferably, *peer* is used in direct relation to P2P while *node* is a more general term.
NVE	Networked Virtual Environment; Large-scale virtual environments like MMOGs or multimedial virtual worlds using a distributed networking approach an in the context of this thesis applied P2P technology.
P2P	Peer-to-Peer; A distributed system where each peer is considered equal and participates on processing the workload.
Pub/Sub	Publish/Subscribe; A distributed mechanism to decouple message composition and delivery.
RTT	Round-Trip Time; The delay of a sent packet and a received response.
SPOF	Single Point of Failure; A Single Point of Failure causes a whole system to malfunction due to a failure of only one component.
TCP/IP	Transmission Control Protocol/Internet Protocol; Nowadays, the most important networking protocol for applications.

List of Figures

1 Introduction

During the last decades, extensive research was done in the area of distributed computing starting in the 1970s. Applications like e-mail or the Web began on getting popular and widely accepted. However, most of these applications used a simple C/S (Client/Server) approach that led to several disadvantages with issues in scalability, availability, or security. Especially the ever-growing number of Internet users is creating more and more distributed applications with a huge number of concurrent users. Obviously, the C/S paradigm encounters many difficulties due to the fact of only one or few central instances and many directly connected clients. Vertical scaling, that is, adding more resources to a single node, may help the server instance to a certain degree (Michael et al., 2007). In this way, it would be possible to add more main memory, more (multi core) processors, or faster storage systems. However, vertical scaling is quite limited in terms of costs and reliability. Horizontal scaling, that is, dividing the work load onto more nodes, provides better scaling. Horizontal scaling includes hardware techniques like load balancing or software driven distributed networking systems (Michael et al., 2007).

Such distributed networking systems can be P2P (Peer-to-Peer) systems. They try to cope with the drawbacks defined above and contribute new possibilities for distributed applications. The conception of P2P is to move the responsibilities away from a single instance towards many equally treated clients. Section 1.1 gives a brief introduction to the different architectures that P2P systems can have. The area of NVEs (Networked Virtual Environments) is introduced in section 1.2.

Another paradigm in distributed computing is Pub/Sub (Publish/Subscribe) which aims to decoupling publishers of messages from subscribers and, thus, increases flexibility and scalability. Such systems are presented in section 1.3.

The goal of this thesis is to combine P2P with Pub/Sub technology in the context of NVEs and to leverage the potential of each of these technologies. The next sections give background knowledge and introduce the different technologies for a better understanding.

1.1 Peer-to-Peer Systems

P2P systems are distributed network architectures where each participating client node – called peer in this context – shares some of its own processing power, bandwidth, and other resources and makes them available to other participants. Hence, P2P applications are suitable for processing large amounts of data with a plethora of concurrent users. Especially in times of ever-increasing bandwidth availability and number of online users, P2P helps to mitigate the negative effects a pure C/S network would suffer. Another important aspect of P2P systems is to save expenses as peers can handle most of the work without relying on a big and costly data center. In particular, many online video streaming or content distribution applications based on P2P technology have emerged. These include BitTorrent DNA[1], Internap[2], and Octoshape[3] for commercial peer assisted content delivery providers and Voddler[4] for video on demand using P2P technology.

As stated earlier, the lack of a central server instance in P2P and the complete independence[5] of single servers demand different strategies of handling all peers. Consequently, the P2P system and its nodes

1 `http://www.bittorrent.com/dna`
2 `http://www.internap.com`
3 `http://www.octoshape.com/`
4 `http://www.voddler.com/`
5 Note, that many systems need a special node for “bootstrapping”, i. e., creating a new network.

have to handle crucial tasks like node detection, message propagation, or failure recovery on their own without the help of a global and all-knowing coordinator. Especially scenarios with a high churn rate put a high strain on the network. The churn rate is defined as the number of joining and leaving peers per time unit. A high churn rate may indicate the outage of a big part of the overlay structure for many, that is, hundreds or thousands of peers simultaneously. The P2P network has then to cope with this problem and update the information stored on each remaining node to ensure proper functionality again. Note, that a P2P network or overlay is different to the real underlying network. Nearby peers in the network layer do not have to be neighbors in the P2P layer and vice versa.

Even with ignoring high churn scenarios, management of the whole network causes work in varying degrees for the underlying P2P protocol. Plenty of different implementations and approaches exist that try to optimize certain properties like bandwidth, latency, reliability, or overhead. However, as there is no general panacea, the right P2P system needs to be chosen according to the application's behaviour and needs.

However, P2P based systems can benefit from resources of all peers in order to achieve good scalability, performance, and reliability. Most notably, distributing load over all peers greatly helps to decrease the load on a single system. Furthermore, dynamically changing workloads can be easily handled by P2P systems as joining peers automatically share their resources. Thus, even at the peak of the workload, requests can be processed. Ideally, the workload scales with the number of peers. Contrary, C/S systems require exact prior planning to be able to handle peak situations. Additionally, in times of a small workload, for example during night hours, data centers still need to operate with the complete server capacity whereas P2P systems automatically adapt to this situation.

Further, the danger of a SPOF (Single Point of Failure) can be eliminated which greatly improves reliability. On the downside, handling of data is more complicated due to the distributed nature and participation of many nodes. Consider a scenario of a distributed P2P backup where the data is replicated among peers. Peer *A* has the data in version *1* and then goes offline, i. e., leaves the network due to a crash or similar reasons. After that, peer *B* updates the data to version *2* before *A* reconnects. *A* now has stored outdated information that needs to be updated to the correct version. Achieving this in a decentralized network of equal peers with no or very limited global knowledge is no trivial task.

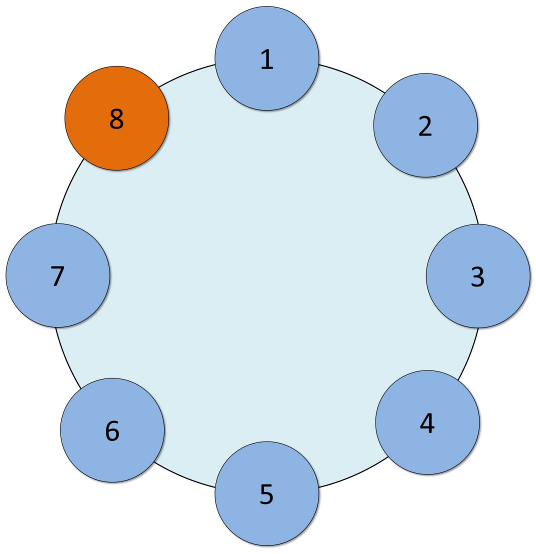
Although many of the implementations of P2P systems share very similar properties, they can yet be distinguished into two types: structured and unstructured. The following sections will present their similarities and differences as well as their pros and cons.

1.1.1 Structured Peer-to-Peer Systems

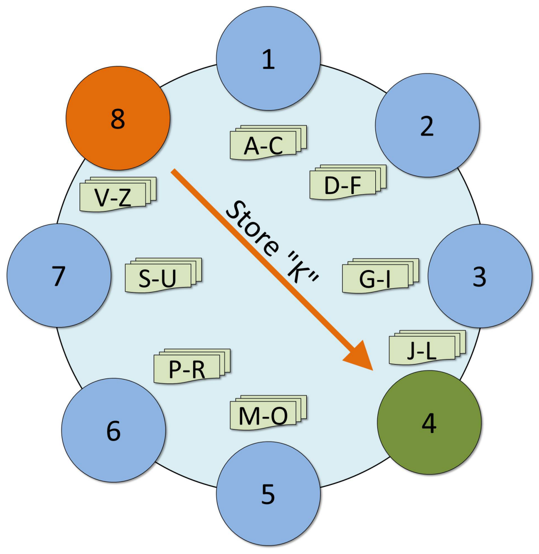
As the name suggests, structured P2P systems organize the peers into a well-defined pattern with the focus on fast key retrieval and routing. That is, they optimize on inserting and retrieving keys on the fastest way possible. Usually, this is done by utilizing hash functions and specific mapping rules. Thus, an overlay is created that obeys the rules and forms a deterministic topology as seen in figure 1.1a. Most structured P2P systems apply DHTs (Distributed Hash Tables) which is illustrated in figure 1.1b. In this example, peer *8* wants to store the data *K* in the system. After applying the hash function *H(x)*, the resulting key is *4*. The request is now routed by the DHT to the responsible peer *4* and stored there.

Note, that every peer is responsible for certain key space of the whole DHT. Thus, peers may become hot spots if there are many items for a small key space (cf. Zipf's law or Pareto principle). However, workarounds exist like dynamically associating the key space for the hot spot area to multiple peers (Aberer et al., 2003) or remapping the designated place in the DHT to another peer.

Searching for data is equivalent. A peer issues a search request that is then routed through the DHT until the peer responsible for the hash is found. As every key has a well-defined place within the DHT searching and key retrieving is quite efficient and fast. Search queries only work well for static keys which



(a) Structured P2P Overlay Network



(b) Storing Data "K" in a DHT

Figure 1.1: Structured P2P Systems

is the downside of DHTs. This means that querying for ranges, conditions, or complex queries including Boolean operators will not work (efficiently). Well-known representatives of DHT based systems are *Chord* by Stoica et al. (2001), *Pastry* by Rowstron and Druschel (2001), and *Kademlia* by Maymounkov and Mazieres (2002).

Another approach are tree-based systems that build an overlay based on a prefix tree also known as *Trie*. Query matching can be done using the prefix and matching it against the tree. Tree based systems can, in contrast to DHT based systems, work on a non-uniform load distribution with balanced peer traffic and enable range queries. Again, complex queries are not feasible. *P-Grid* by Aberer et al. (2003) is a representative of this approach. This system attempts to combine both the advantages of a DHT with advantages of unstructured P2P systems.

1.1.2 Unstructured Peer-to-Peer Systems

Contrary to structured P2P systems, unstructured systems do not organize their peers according to a pattern. This implies that peers are randomly connected to other peers. Hence, searching for data is not as efficient as in structured systems as no direct relation between peer and key is present. Furthermore, data is most likely retrieved by flooding the network. That is, asking every peer for the data in the whole network which is achieved by sending a search request to all connected peers which in turn forward the request to their connected peers. Therefore, the results are quite reliable and searching can be accomplished with an adequate latency. Obviously, this creates a huge amount of traffic that needs to be coped with. However, unstructured systems allow arbitrary complex search queries as answering of the request is done directly by the peer and not by the in flexibility limited overlay. In contrast to structured systems where the query needs to be transformed into key-value pieces, unstructured systems allow queries in any complexity. This includes the use of Boolean arithmetic, range queries, hierarchical selections, and even full-text search. Concluding, any form of selective queries is possible as long as the peers are powerful enough to process these. Another related advantage is that the application can make use of already available query libraries like *SQLite* or *XQuery*. In contrast to structured systems, unstructured systems can leverage this existing potential as there is no need to design another query language that fits together with the overlay. For that reason, unstructured systems are more appropriate for applications that operate on dynamic data that might be queried in various ways. An example would be a full-text search based on a distributed and collaborative Wiki. Another important aspect is the built-in load balancing. While DHT based systems suffer severely from hot spots in non-uniform distributions — natural languages show an Zipfian load distribution — unstructured systems do not express this weakness due to their random distribution of data onto the peers. The impact of such a distribution in structured P2P systems causes the peer which is responsible for the biggest hot spot a load which is many times higher than average. However, this peer only has an average capacity available and will thus suffer from overload. Subsequently, this will cause the network to act more slow or even make that key space of a DHT overlay inaccessible.

According to Kang et al. (2010), unstructured P2P systems can be differentiated into superpeer, centralized and decentralized networks. Further, hybrid approaches combining both superpeer and decentralized techniques exist.

- *Centralized* networks use a C/S approach for managing the overlay and searching. Only the transfer of data is done directly between the peers. This may be the most simple P2P approach but still saves bandwidth and enables fast searching at the cost of decreased robustness and high vulnerability to SPOFs. The most popular centralized system was *Napster* with millions of users at its peak.

- In terms of network robustness and scalability, *decentralized* networks have an advantage. *Gnutella* was one of the most famous decentralized unstructured systems[6]. *BubbleStorm* is another decentralized system which will be presented in detail in section 2.2.

- Finally, *Superpeer* networks also achieved a high popularity through programs like *KaZaA* or *Skype*.

As mentioned before, flooding is quite inefficient especially with an increasing number of peers in any of the unstructured systems. Several methods exist to mitigate the unwanted effects of flooding including worse scalability, slow response time, and high network traffic. These methods include *restricted flooding*, *superpeer* networks, and *random walk*s.

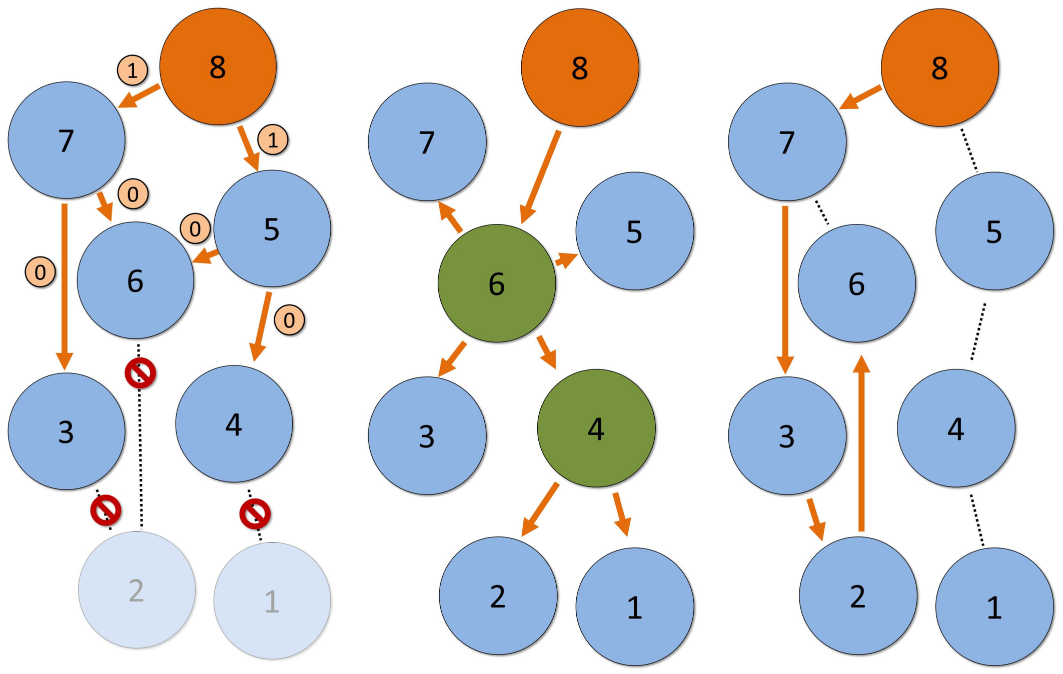
Restricted flooding is the most basic approach. Instead of sending and forwarding one search request to every connected peer, each request times out after a defined time or number of hops. Thus, flooding is restricted to a certain perimeter around the node, which – in turn – decreases search quality severely. Further, data that is outside of this perimeter cannot be found although it may be present in the network. Concluding, restricting the flooding cannot be seen as a proper approach for most implementations. Restricted flooding is illustrated in figure 1.2a. In this example, peer *8* sends a search query to all its connected neighbors with a maximum hop count of two. After peers *5* and respectively *7* receive the search query, they decrease the hop count by one. Here, the query ends at peers *3*, *4*, *5*, and *6*. Another problem can be seen in this example that is valid for all unstructured P2P systems: cyclic connections. These cycles are especially problematic for flooding as a packet is routed over the same links multiple times (peer *6* in the example). However, restricted flooding does not suffer great damage from this behavior as the hop count is decreased steadily. Yet, coping with or better avoiding cyclic connections is an issue of the architectural design and implementation.

Superpeer networks take the first step towards reduced traffic and faster response times without such strict flooding restrictions. In a superpeer network two kinds of peers exists, the normal peers and the superpeers. A superpeer is a peer that has more available resources, especially bandwidth, but also processing power. After a peer is promoted to a superpeer, it will be responsible for a larger number of normal peers. Normal peers can no longer directly communicate with other distant peers — communication is done using the superpeers. If a normal peer then starts a new search query, it will be routed to

[6] At the time the protocol was in version 0.4.

the superpeer which in turn forwards the query to other directly connected superpeers. Every superpeer will then ask their peers for a result which, in case the wanted data is found, is routed back. Additionally, superpeers can cache the meta information about their peer's data to improve responsiveness. However, in a very large network even the number of superpeers will exceed a certain threshold which makes this approach again badly scalable. A solution would be to limit the number of queried superpeers (cf. restricted flooding) or to structure superpeers hierarchically. Further, the process of promoting a peer to a superpeer is difficult in terms of detecting sufficient and stable resources or even unwanted by the user. A superpeer network is illustrated in figure 1.2b. As explained before, only the superpeers *4* and *6* communicate directly with the peers they manage. If a request is sent from peer *8*, it will be routed to its superpeer *6* which forwards the query to its other children as well as the connected superpeers to get replies.

*Random walk*s provide a better solution for restricted flooding of the network. To enable this, statistical algorithms are applied that can ensure high reach rate for the network overlay. For example, instead of flooding all connected neighbors, a query is sent to a randomly selected neighbor. This neighbor then checks for matches and forwards the query to another randomly selected neighbor until the query times out. Using this approach results in an increased diversification in selected peers and thus in a most likely better query result as well as a more balanced link load. However, as this is a random approach, it is possible that the selected neighbors are still in a close perimeter which does not contain the wanted data. Moreover, a random walk can increase latency severely. The reasons are long distances that might be traveled and the serial nature of execution. An example for a random walk is illustrated in figure 1.2c. Here, node *8* starts a query and thus forwards it to a randomly selected neighbor (*7* in this case). Peer *7* then selects another random neighbor and so on until the query times out or results are found.



(a) Restricted Flooding with a Maximum Flooding Depth of 2 Hops **(b)** Superpeer Network with Superpeers *4* and *6* **(c)** Random Walk

Figure 1.2: Unstructured P2P Techniques

All presented techniques suffer from one issue which cannot be left out of consideration: Sent queries are not guaranteed to receive a complete or even any result set. Therefore, more advanced techniques have to be applied as seen in section 2.2.

A concluding comparison of major properties of structured and unstructured P2P systems can be found in table 1.1.

Property	Structured	Unstructured
Joining	easy to complex	easy
Churn Recovery	complex	good
Scalability	good	good
Placement of Data	defined place	randomly
Query time	fast	slow
Query accuracy	good	bad
Query Complexity	simple	complex
Query Language	one / defined by overlay	any / defined by application
Load Balancing	prone to hot spots	built-in

Table 1.1: Structured vs. unstructured P2P systems

1.2 Networked Virtual Environments

NVEs are applications that feature a virtual world where entities can roam and interact with each other. Further, these entities can be physically located anywhere on the globe as the whole virtual world is a distributed networking system. In this thesis, when speaking of NVEs, P2P technology is the approach to connect the entities in the virtual world and perform important tasks like interest management. NVEs are found mainly in the area of computer gaming or virtual (social) worlds. During the last decade, NVE applications — notably MMOGs (Massively Multiplayer Online Games) — increased their popularity due to progress in available client resources and bandwidth. Especially MMOGs with a plethora of simultaneous players put high demands on the back-end system that handles players and game world states. Recent MMOGs like *World of Warcraft* with approximately 11.5 million players worldwide utilize an infrastructure of about 13,250 server blades, 75,000 processor cores, and 112.5 Terabytes of blade main memory (Miller, 2009). This tremendous amount of required processing and storage power hence gobbles up huge amounts of money. Other drawbacks of traditional C/S architectures include the risk of a SPOF and limited scalability (Neumann et al., 2007).

For these reasons, sharing the workload to all participating clients seems advantageous. Nevertheless, constructing a MMOG solely on a P2P network layer entails several challenges that need to be dealt with.

Crucial for a positive user experience in real time games are minimal delays between user-initiated actions that include movement in the virtual world as well as interaction with other objects and players. Particularly critical are very low delays in FPS (First-Person Shooter) games for the fast-paced game play while MMOGs (mostly the genre of role playing games) allow delays to be higher to some degree as game updates are less frequent in general (Knutsson et al., 2004). Reducing overall delay is difficult as the basic networking delay can hardly be modified. Nevertheless, reducing the delays caused by the P2P overlay or the application is possible. For instance, the routing could be optimized so that the message RTT (Round-Trip Time) drops. Other techniques include dead reckoning where the movements of a player are predicted based on their previous movements. After the real movements have been received, the predicted state is corrected (Neumann et al., 2007).

As client bandwidth is quite restricted in most cases, there is the need to keep bandwidth utilization to a minimum. Especially for frequently sent game state updates, elaborate techniques like interest management or delta encoding have to be applied (Pang et al., 2007). Interest management handles the player and the surrounding objects that are possible for interaction. Often, AOI (Area of Interest) techniques are used that try to narrow down message traffic through careful consideration of other interactive objects close by. The player then sends only updates to others, that are in the AOI. Although AOI works well in most cases, there are exceptions. Interest heterogeneity is such a case, when a few players are very interesting to many others. A sample scenario could be a flag-carrier in a FPS multiplayer game, which all opposing players hunt. The flag carrier then has to send updates to all pursuers which will lead to large bandwidth requirements. To cope with this issue, multicast techniques could be deployed (refer to section 7). Further, when sending updates, delta encoding can help to reduce message sizes. Delta encoding only sends the changes between consequent updates instead of a complete one.

Another important aspect is the distributed management of the game state, that includes all information about players and objects in the game world. In contrast to server based management, replication has to be done involving many peer nodes to ensure reliability and availability. In this context, the handling of game state updates are of great importance for the consistency of the whole game world. Again, the mechanism to update game states should need as few messages as possible to keep peer traffic low.

There are even more challenges in the area of networked virtual environments like security (especially cheating protection), consistency, and reliability (Hu and Liao, 2004). They are, however, beyond the scope of this thesis and will not be explained further.

1.3 Publish-Subscribe Systems

Due to the increasing complexity of today's applications, the typical architecture changed from being on a single machine to being distributed across many computers. As program requirements became even more complex, independent parts of the application were outsourced to separate systems. Nowadays applications often utilize dozens to thousands of single computers that interact with each other. These systems are using synchronous communication and point-to-point connections in most cases. As a result, new problems like limited scalability, fragile reliability, or restricted flexibility arise. Further, this approach led to difficult large-scale application development as the aspects of communication were dominating.

To cope with the issues mentioned, Pub/Sub systems have been developed. The main idea behind Pub/Sub is, to loosely couple the whole system using a messaging service component (or event notification service). This is achieved by decoupling the creators of messages, referred to as publishers, from the consumers of these messages, referred to as subscribers. Publishers and subscribers do not require any direct mutual knowledge and act independently. A publisher offers a service for which subscribers can express their interest in. In case of new data, a new message is created by the publisher and propagated automatically and asynchronously to all interested subscribers via the event notification service. Figure 1.3 shows a general Pub/Sub system and its components.

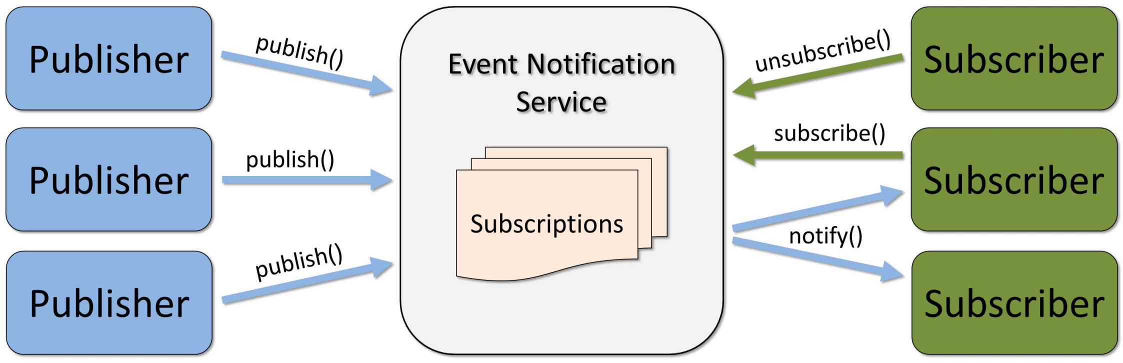
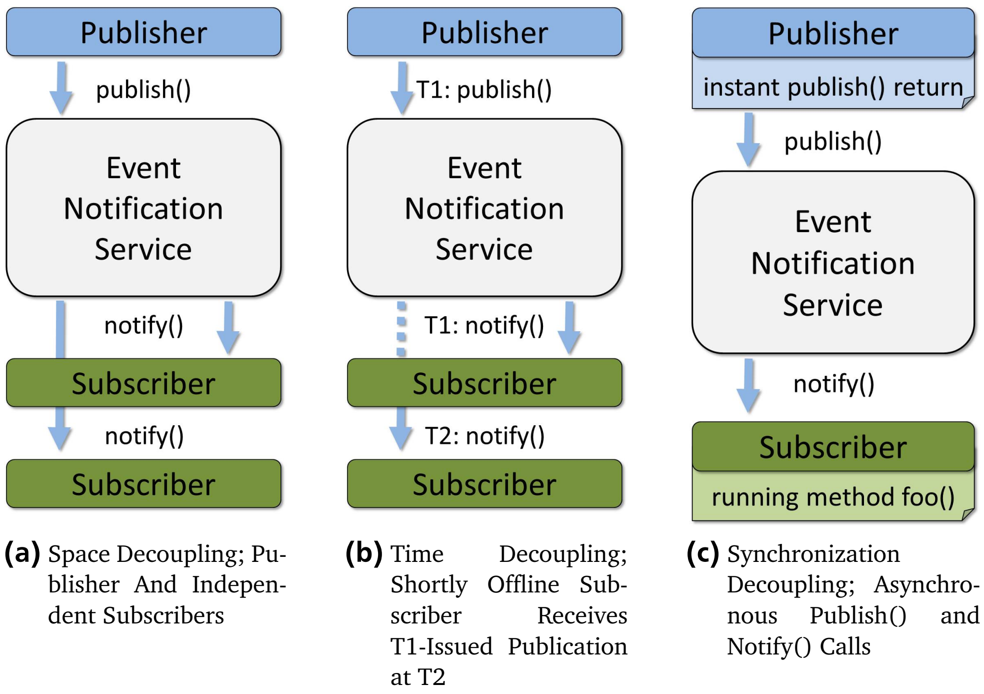


Figure 1.3: General Pub/Sub System and Their Components

In general, Pub/Sub decouples in space, time, and synchronization (Eugster et al., 2003). Space decoupling (figure 1.4a) implies the independency of both publisher and subscriber. Further, both participants do not hold any references to each other and subscribers receive messages indirectly and have no knowledge of other subscribers. In addition, publishers and subscribers are decoupled in time as they do not have to be connected at the same time (figure 1.4b). The event notification service handles queued requests and sends them after the other node is available again. Last, communication is asynchronous at both publisher and subscriber, meaning that no blocking calls are made (see figure 1.4c). Publishers can continue with their work immediately after they send a message and subscribers get notified using a callback for incoming messages.



(a) Space Decoupling; Publisher And Independent Subscribers

(b) Time Decoupling; Shortly Offline Subscriber Receives T1-Issued Publication at T2

(c) Synchronization Decoupling; Asynchronous Publish() and Notify() Calls

Figure 1.4: Types of Decoupling in Publish/Subscribe Systems

Pub/Sub systems can be categorized into three major types (Eugster et al., 2003; Shen et al., 2010):

1. topic-based,
2. content-based, and
3. type-based.

Topic-based Pub/Sub systems embody the most basic form of Pub/Sub. As the name implies, subscribers sign in for a specific topic. Thus, every unique topic creates a new subscription group where publishing an event can be seen as broadcasting a message into the group. Improvements of topic-based systems include hierarchical subscription or wildcard matching. Hierarchical subscription allows nested topics, i. e., a general topic like *Weather* for weather information. This topic could include a sub topic like *Europe* which in turn could include *Germany*. Hence, a fine-grained subscription management based on the subscribers interests is possible.

Another form of Pub/Sub are content-based systems. In contrast to topic-based systems, where interest is expressed through subscribing to a specific topic, content-based systems do a filtering of messages based on the actual message content. This increases the expressiveness in comparison to topic-based systems but also introduces more complex processing. Each message can be filtered using a definition of the filter expression for a given field in the message. In detail, comparison operators like (<, ==, >) can be used to define constraints. To create even more complex filters, several expressions can be combined with Boolean logic (and, or, . . .). An example would be the filter `"country == 'Germany' AND temp > 25.0"` for all German cities whose temperature is above 25 degrees centigrade.

Type-based Pub/Sub systems are based on the observations made in topic-based systems, namely the similarities in structure and content of the published messages. Therefore, subscriptions are no longer topic-based, but rather based on their type (Eugster et al., 2003). These types can have a hierarchy just like in object-oriented languages including checking of type safety at compile time. A subscription is then made by registering with a specific object data type at the messaging service.

All three mentioned types of Pub/Sub systems need to be able to disseminate messages which publishers emit. The Pub/Sub paradigm does not imply specific rules about that matter. Different approaches exist to spread the message to the subscribed participants: mainly point-to-point communication or multicasting techniques. However, the effectivity of disseminating messages greatly depends of the cost of filtering subscriptions (Eugster et al., 2003).

In general, Pub/Sub have an event notification system that acts as mediator between the event producers (publishers) and consumers (subscribers). Publishers and subscribers then send their messages to this event system which in turn is responsible for correct delivery of the messages. Many different routing mechanisms exist which disseminate messages to peers. Popular techniques include event flooding, subscription flooding, filter-based routing, or rendezvous routing. Obviously, flooding techniques flood the network with the data. Filter-based routing creates a multicast tree based on existing subscription filters. Rendezvous routing is based on subscription and publication functions. Given a distributed subscription, a publication is forwarded successfully if subscription and publication match on the same node (rendezvous node).

1.4 Summary and Structure of this Thesis

This introduction has shown that by leveraging P2P technology, a solution for the issues of limited scalability and robustness in traditional C/S architectures exists. However, each of these approaches have their own assets and drawbacks. Usually, structured P2P systems suffer from being unable to do performant queries while unstructured systems have issues with fast key retrieving or routing. Pub/Sub systems have been introduced that help to decouple message publishers and subscribers and enable solutions for interest-targeted message delivery.

Combining P2P with the Pub/Sub paradigm in the context of NVEs will be the central aspect of this work. Therefore, the MMOG game Planet π4 (section 2.1) will be used as the NVE. As mentioned earlier, the handling of the AOI of each player is of great interest. For that reason, it is important to quickly detect all neighbors in close proximity. As the NVE supports a large amount of concurrent players it is not feasible to rely on the classical C/S pattern. Therefore, P2P technology is used to cope with the aforementioned drawbacks of C/S systems. As every player is surrounded by other entities (that may be other players or game objects) only entities that are in the visible range are of interest. Thus, a spatial Pub/Sub approach is chosen where peers can express their interest in neighboring peers within that spatial area. This approach is similar to a topic-based system with the opportunity to additionally filter by coordinate area in an Euclidean space. For this to work the *BubbleStorm* P2P overlay (section 2.2) is utilized. Section 3 will go into more detail about the concept of the Pub/Sub overlay. As updates of the position of each player are plentiful and of a fire-and-forget connection strategy[7], the *CUSP* transport protocol (see section 2.3) is used instead of mere TCP/IP communication. Subsequently, the implementation is presented in section 4 with an in depth coverage of all components. Section 5 will present the evaluation setup, the gathered results, as well as a comparison with other networking approaches. The thesis is concluded with section 6 and the prospect of further work in section 7.

[7] For an update a connection is opened, the new position is transmitted and then closed again.

Summarizing, this thesis will answer three crucial questions:

1. **Is a Pub/Sub approach feasible in NVEs?**
   To be able to answer this question, the NVE application Planet $\pi4$ is used. The Pub/Sub system will be integrated to Planet $\pi4$ as a networking engine and will handle interest management besides other networking functionality. An evaluation of this approach will show, if performance is appropriate for fast game play with many concurrent players.

2. **Can the Pub/Sub overlay be built on top of an unstructured P2P system in order to achieve satisfying results?**
   BubbleStorm is an unstructured P2P system that uses a probabilistic approach for data dissemination. This thesis will determine, if such an approach is feasible and if interest management accuracy can be assured.

3. **How does the developed system perform and how does it compare to other approaches?**
   To measure performance of the developed overlay, a discrete event simulation will be performed. Additionally, this work's implementation will be compared to the unstructured P2P overlay pSense. Therefore, important metrics will be defined for a profound comparison.

2 Related Work

This section introduces *Planet* π4 (section 2.1), the game used to evaluate the P2P based Pub/Sub overlay, and *BubbleStorm* (section 2.2), the underlying P2P system. Additionally, *CUSP* is presented which will serve as the networking replacement for pure TCP/IP for data transfer between the peers (section 2.3). P2P systems that take spatial proximity into consideration are discussed in section 2.4. P2P systems that include Pub/Sub functionality are presented in section 2.5.

2.1 Planet π4

Planet π4 is the MMOG which is used to evaluate the implementation of this thesis. Planet π4 was originally developed for the *SpoVNet* ("Spontaneous Virtual Networks") project[1] at the University of Mannheim by Triebel et al. (2008) in order to demonstrate P2P-based network overlays. Since then, Planet π4 has been developed further to improve game play and to adapt to other overlays like *pSense* (cf. section 2.4.1). This work extends Planet π4 by the possibility to use the P2P system BubbleStorm as an interest management system for players. As the aim is to measure the efficacy of interest management of the newly developed P2P-based Pub/Sub approach Planet π4 is suitable.

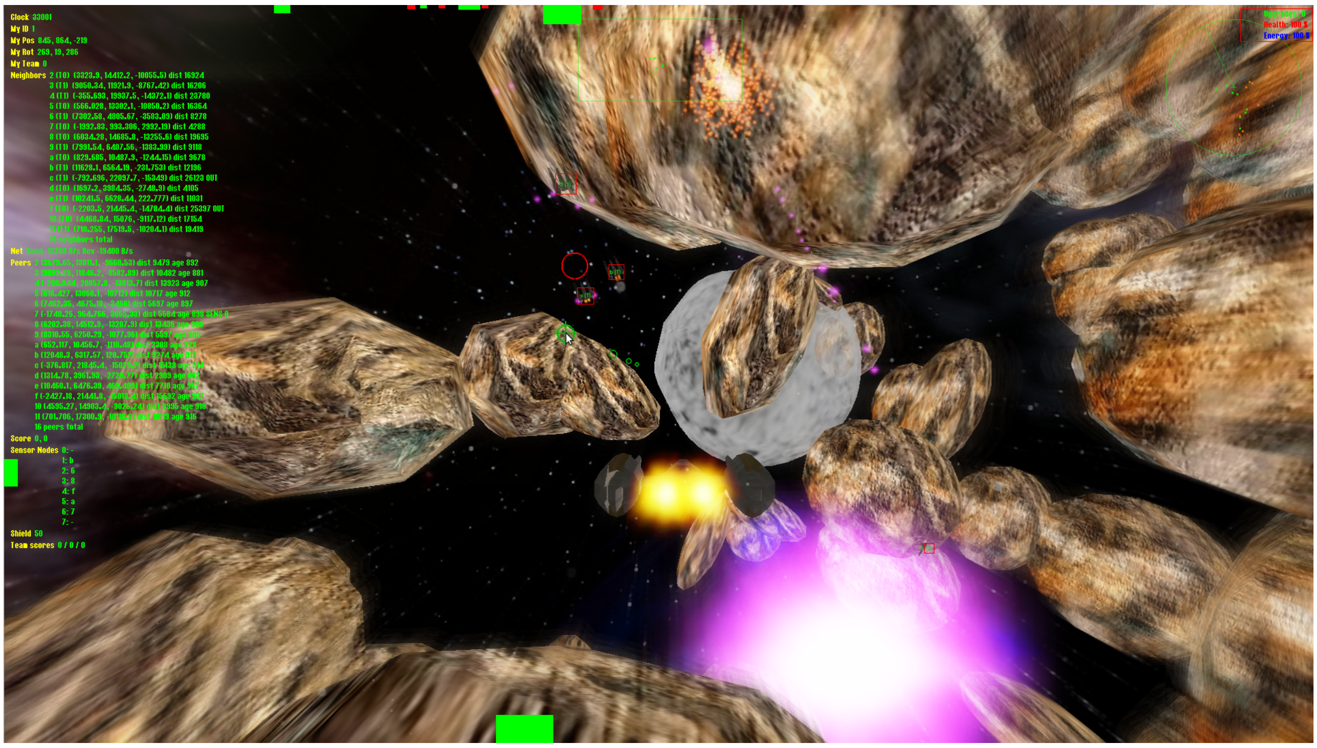
MMOGs like Planet π4 support plenty of concurrent players which are able to interact with each other. Typically, for a game in the space vehicle combat setting, each player controls one spaceship that can freely roam through the game world. Each ship is equipped with a weapon that can shoot energy beams to hostile players. When a player is hit by such a beam it will decrease the available shield value. If the shield value reaches zero the ship will explode and the player's ship will respawn at a different location. Additionally, Planet π4 features team-based combat as there are two or more teams that play against each other. As previously mentioned, Planet π4 was created to evaluate different network architectures and measure how they affect game play. To get first hand results from human testers who assess the game, it is necessary to have an appealing and interesting way of playing. For that reason, plenty of changes were made to the original Planet π4 version. First, each ship has an installed speed booster that can increase the velocity greatly for example to escape from an enemy or fly to a shield regenerator. Second, to make maneuvering more demanding, the game world is filled with asteroids as obstacles which the player has to dodge. Hitting an asteroid or other obstacles will reduce the ship's shields and eventually let the ship explode. Third, the player gets the possibility to repair the ship by flying into a special zone that refills the shields. Further, weapon upgrades and more are planned to improve game play.

Planet π4 features real-time characteristics and fast-paced game play. Especially the characteristics of having an infinitely big and explorable game world will stress the overlay. Further, no boundaries like in other games exist, that is, there are no rooms or areas that separate players with a different AOI. This makes handling of Pub/Sub more demanding. Additionally, a fast game play implies plenty of player's movement and thus stresses the overlay which is responsible for player recognition and position updates. Further, as implementations with different P2P overlays already exist in Planet π4 it is suitable for evaluation and comparison.

Figure 2.1 shows a screenshot of a running instance of the game. The player's space ship is in the center of the screen surrounded by asteroids and other players.

Another important property of this version of Planet π4 is that it can be simulated using a discrete event-based simulator. Therefore, it not only provides a normal real-time mode but further offers a

[1] `http://www.spovnet.de`



```
Clock 33001
My ID 1
My Pos 845, 864, -219
My Rot 269, 19, 286
My Team 0
Neighbors 2 (T0) (3323.9, 14412.2, -10055.5) dist 16924
          3 (T1) (9050.34, 11921.9, -8767.42) dist 16206
          4 (T1) (-355.693, 19937.5, -14372.1) dist 23780
          5 (T0) (566.028, 13302.1, -10850.2) dist 16364
          6 (T1) (7302.58, 4805.67, -3583.09) dist 8278
          7 (T0) (-1992.83, 993.306, 2992.19) dist 4288
          8 (T0) (6034.28, 14685.8, -13255.6) dist 19695
          9 (T1) (7991.54, 6407.56, -1383.99) dist 9118
          a (T0) (829.685, 10487.9, -1244.15) dist 9678
          b (T1) (11628.1, 6564.19, -231.753) dist 12196
          c (T1) (-792.696, 22097.7, -15349) dist 26123 OUT
          d (T0) (1697.2, 3984.35, -2748.9) dist 4105
          e (T1) (10241.5, 6628.44, 222.777) dist 11031
          f (T0) (-2203.5, 21445.4, -14784.4) dist 25397 OUT
          10 (T0) (4468.84, 15076, -9117.12) dist 17154
          11 (T1) (710.255, 17519.5, -10204.1) dist 19419
          16 neighbors total
Net Send -15288 B/s Rcv -19400 B/s
Peers 2 (3579.65, 13811.1, -9660.53) dist 9479 age 892
      3 (9092.29, 11846.2, -8582.89) dist 10482 age 881
      4 (-208.944, 20057.8, -13813.7) dist 13923 age 907
      5 (616.427, 13060.1, -10712) dist 10717 age 912
      6 (7462.35, 4675.18, -3460) dist 5697 age 897
      7 (-1740.26, 964.766, 3065.33) dist 5584 age 898 SENS 0
      8 (6202.38, 14512.9, -13207.9) dist 13436 age 906
      9 (8310.55, 6250.29, -1077.96) dist 5597 age 905
      a (652.117, 10456.7, -1110.48) dist 2308 age 928
      b (12048.3, 6317.57, 120.752) dist 9274 age 912
      c (-376.817, 21845.4, -15020.2) dist 15138 age 914
      d (1314.78, 3961.93, -2728.77) dist 2909 age 882
      e (10460.1, 6476.39, 463.409) dist 7710 age 967
      f (-2427.18, 21441.8, -15018.4) dist 15692 age 907
      10 (4595.27, 14903.4, -9025.24) dist 8995 age 918
      11 (701.706, 17300.9, -10118.8) dist 10119 age 916
      16 peers total
Score 0, 0
Sensor Nodes 0: -
             1: b
             2: 6
             3: 8
             4: f
             5: a
             6: 7
             7: -
Shield 50
Team scores 0 / 0 / 0
```

Figure 2.1: In-game Screenshot of Planet π4

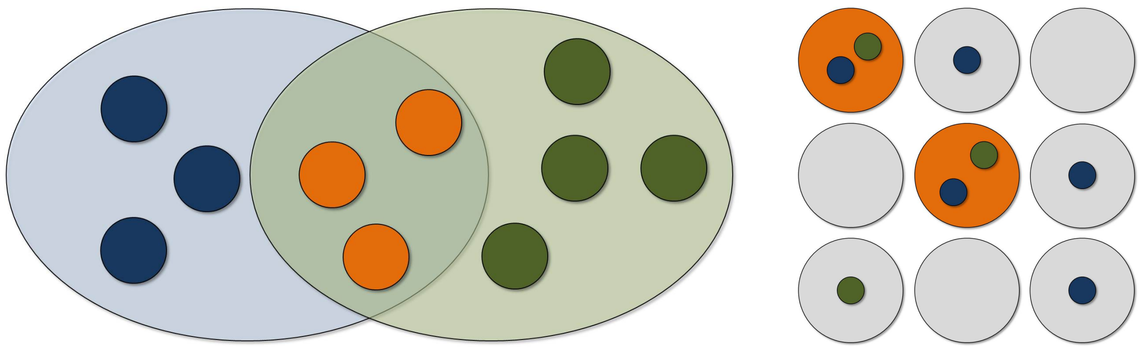
simulation mode. In this mode, Planet π4 and all components including the network overlay are handled by a discrete-event simulator that provides deterministic and thus repeatable simulations. Section 5 provides more information on this topic.

2.2 BubbleStorm

Another fundamental part of this thesis is the unstructured P2P system BubbleStorm by Terpstra et al. (2007a). Like mentioned in section 1.1.2, unstructured P2P systems excel in offering the possibility to enable a more flexible search. BubbleStorm therefore focuses on that aspect and leaves the implementation of the query evaluator to the application programmers. The reason is to support dynamic queries that can match against all documents. More specific, that means, that BubbleStorm offers the network overlay to propagate and route a search query but lets the application handle the actual reply. Further, the BubbleStorm overlay can probabilistically assure an exhaustive search within given parameters. Besides, BubbleStorm offers good scalability with a fast search. Terpstra et al. (2007b) state that a small query takes less than one second to reach a single document in a network with one million nodes. Furthermore, BubbleStorm features load balancing to avoid hot spots on single nodes, load distribution to adapt to heterogeneous bandwidths, and can handle high churn rates of peers.

Contrary to other unstructured P2P systems, BubbleStorm not only replicates the data onto several peers but also the queries. There are roughly $O(\sqrt{n})$ copies in a network with n peers at the same time. Figure 2.2a illustrates this concept with a blue *data bubble* and a green *query bubble*. Bubbles are an abstraction for parts of the network containing a subset of peers. In this figure, obviously, both bubbles overlap and the formed intersection is called *rendezvous*. As the replication of data and queries is done pseudo-randomly, there are peers that are only in the data bubble, only in the query bubble, or in both bubbles. Peers which are located in both bubbles are the aforementioned rendezvous nodes. Figure 2.2b shows this in more detail. Statistically, the rendezvous nodes follow the *Birthday Paradox*. In short, it concludes that on a randomly chosen set of people one pair will have the same birthday to a certain and unexpectedly high probability. Transferring this to the concept of bubbles leads to the rendezvous nodes. Accordingly, searching for something in the overlay results in creating a query bubble that is placed pseudo-randomly on peers. Nodes that have the required data and receive the query (i. e. the rendezvous nodes) will reply.

Further, the size of the bubbles can be adjusted. In this way, powerful peers can increase the size of their bubble by receiving more traffic and thus relieving the resources of weaker peers. This is achieved by increasing the node degree and thus the outgoing connections. Weak peers however keep less connections resulting in a smaller bubble.

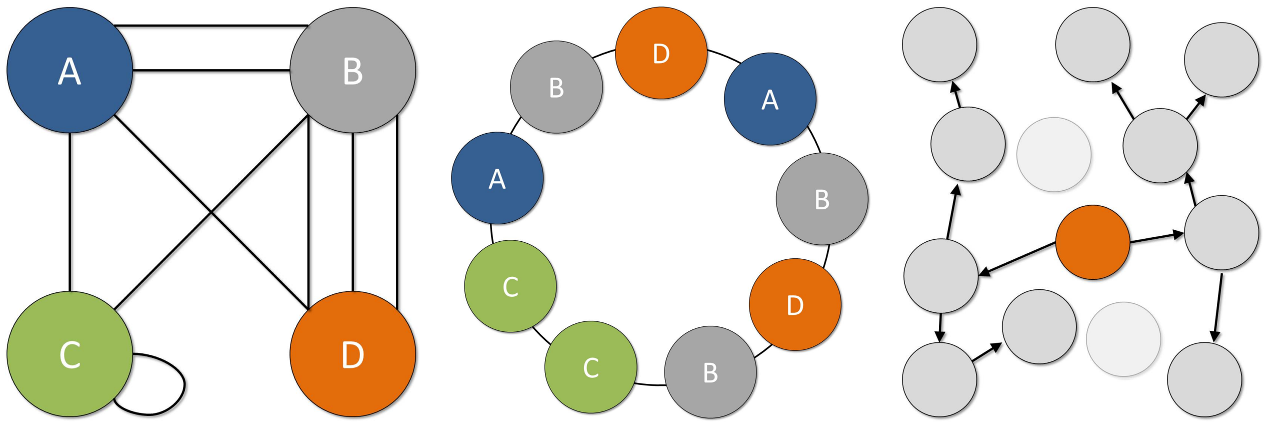


(a) Data (Blue) and Query (Green) Bubbles With Rendesvouz Nodes (Orange)

(b) Placing Data (Blue) and Queries (Green) on Nodes (Gray) for Matching (Orange)

Figure 2.2: BubbleStorm's Concept

More precisely, BubbleStorm creates a random overlay that is based on *Random Multigraphs* as seen in figure 2.3a. They have advantageous properties which make maintaining the overlay easier. First, random graphs easily support the birthday paradox by having randomly selected edges. Following an edge leads to a randomly selected peer. Thus, managing and creating bubbles (i. e. subsets of edges) is very easy. Second, each node has a node degree that is in relation to its available bandwidth. Thus, random walks will follow the edges with equal probability and eventually help to have a better utilization of nodes in heterogeneous networks. Further, the overlay forms an *Eulerian Cycle*, that is, an undirected graph where each edge is visited only once. An Eulerian Cycle makes traversing the overlay possible. Relate to figure 2.3b for an example that shows how such a cycle can be formed out of the random multigraph.



(a) Random Multigraph

(b) Eulerian Cycle

(c) BubbleCast

Figure 2.3: BubbleStorm Overlay

As explained in section 1.1.2, flooding in unstructured P2P systems is inefficient in terms of resource utilization. Therefore, BubbleStorm uses a new primitive called *BubbleCast*. BubbleCasts try to combine the advantages of both flooding and random walks while keeping the downsides low. In detail, a BubbleCast works by executing a random walk in a massive parallel fashion (Terpstra et al. (2007b)). Therefore, the originating node emits a BubbleCast to randomly selected edges. Every selected edge respectively node will then randomly choose new edges. Each BubbleCast has a counter which specifies the number of replicas to create, that is, the number of nodes to reach. After new edges are chosen the counter value will be split on the selected edges. This approach limits the flooding of the network similar to restricted flooding (c. f. section 1.1.2). Figure 2.3c illustrates the concept. Concluding, the advantages of a BubbleCast are low latency, reliable search, balanced link load, and precise node count. As a result, BubbleCasts are used for replicating the data and queries in the BubbleStorm overlay.

Terpstra et al. (2007a) state, that BubbleStorm can survive network crashes of 50 % and a churn of 90 %. This makes BubbleStorm very resilient as the protocol only needs a short time to recover.

Accordingly, the BubbleStorm overlay is chosen as the base of the P2P overlay. The Pub/Sub system developed for broadcasting position updates will be created on top of the BubbleStorm overlay.

2.3 CUSP

CUSP (Channel-based Unidirectional Stream Protocol) by Terpstra et al. (2010) is another building block which is used in this thesis. Typically, message transfer between communicating nodes is handled by the widely used *IP* (Internet Protocol) protocol suites *TCP* (Transmission Control Protocol) and *UDP* (User Datagram Protocol). However, both protocols have some disadvantages that restrain performance when used in P2P or other volatile networks. Especially short-living communication creates a notable overhead in network latency due to the three-way handshake in TCP. The frequently used pattern of opening, sending data and closing the connection thus gets inefficient. UDP can, however, mitigate this drawback but at the cost of reliability, and packet ordering. CUSP tackles this difficulty by combining the advantages of both TCP and UDP and is particularly designed for P2P applications. As explained in section 1.1, P2P puts demands on the networking protocol as communication is short-living, dynamic, asynchronous, and complex in terms of many participants. Further, Pub/Sub systems usually send messages whenever the publisher has new data which can happen at any time. Thus, keeping the connections open to all subscribers puts an unnecessary strain on all nodes.

For both P2P and Pub/Sub systems, CUSP can be used to take advantage. It is a unidirectional stream protocol that operates on so called *channels*. A channel is the lowest entity of connectivity and links two nodes in a low-level fashion. On top of the channels an arbitrary number of configurable streams can be opened. All streams are multiplexed inside the channels and have their own settings. Further, each stream has an assigned *service ID* to easily determine which stream has new data available from the application's point of view. Hence, networking applications can leverage from CUSP's potential by creating streams for every communication purpose.

CUSP is built on top of the UDP protocol to reuse existing router functionalities like *Network Address Translation* (NAT) traversal without necessary changes. Contrary to UDP, CUSP offers cryptographic, reliable, and congestion adjustable connections with lowest latency in mind. For example, the necessary connection setup for a channel requires only one round trip and already includes encryption setup if requested. After a channel is established, streams can be opened without the need of an extra round trip to setup. Thus, streams do not suffer from typical TCP issues like slow start and congestion. It is even possible to open a stream, send data and close the stream in one single packet. This is especially helpful in a P2P scenario where no direct reply is anticipated as for instance in an update message. Further, the often occurring case of broadcasting data to a number of nodes usually requires no acknowledgement to be sent back. Additionally, CUSP offers the possibility to prioritize streams. This means, that every stream can have its own priority in order to separate between very important and less important messages. Thus, data on a stream with high priority will be transmitted earlier than data on a low priority stream. Admittedly, high-priority streams which send data continuously can cause low-priority streams to be blocked indefinitely. Ultimately, handling plenty of simultaneous connections is another P2P typical peculiarity. If connections are opened and closed quickly, TCP usually suffers from keeping the connection in a wait state[2] to guarantee proper reception of the last acknowledgment message. Obviously, keeping the connection like this and maintaining state information is unwanted. Closing a connection in CUSP, however, does not require such waiting states and is more acceptable for P2P applications.

In this thesis the CUSP protocol is used for the direct communication between the peers. The latency of the most likely short-living connections can be minimized as streams are lightweight. Further, broadcasting messages to all peers in the often changing AOI is too much overhead when applying the TCP protocol. Although the use of UDP would be possible, features of CUSP like in-order delivery and reliable sending are deemed advantageous.

2.4 Related P2P Systems

This section first introduces P2P systems that focus on spatiality in a two-dimensional space. The list of notable related work includes *VON*, *GP3*, and *MOPAR*. At first, the system *pSense* is presented which is of most relevance in this section as this thesis' spatial Pub/Sub implementation is evaluated with it.

2.4.1 pSense

Schmieg et al. (2008) present the unstructured P2P overlay *pSense* which aims at precise neighbor detection including dissemination of position updates in the area of NVEs. Further, pSense uses a multicasting technique for message delivery. Basically, pSense separates the area of the player's AOI and the distant space. This area outside of the AOI is partitioned into *sectors* and monitored by *sensor nodes*.

Every node in pSense manages two lists in order to keep track of the network and prevent network partitions. The *near node list* contains the *near nodes*, that is, nodes which are in the AOI of the node. These entries get updated as soon as position updates are available. Schmieg et al. (2008) state that this list is up-to-date only to a certain degree as fast moving nodes may be missed at one time or entries that are already out of visibility range are contained. The second list is the *sensor node list* which contains nodes that are outside of the AOI to a certain extent. These sensor nodes monitor the more distant space in order to prevent network partitions. In case of distant nodes coming closer these sensor node will inform the near nodes of the new incoming node. Then, the near node can include the new one in its near list if the AOI matches.

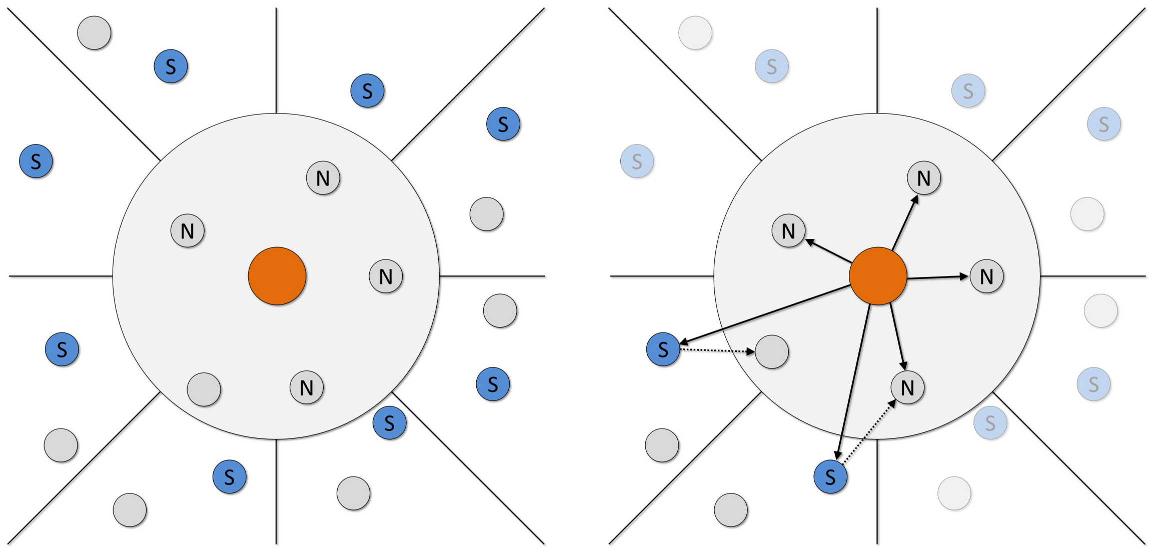
For position dissemination pSense utilizes a localized multicast. When a node changes its position and wants to inform other nodes (both near and sensor nodes) the update will be sent directly. However, as the update message may not be received by all neighbors that are interested forwarding is applied. More precisely, a set of planned receivers with both near and sensor nodes is created. If this list contains too many entries – which would create a too high outgoing bandwidth demand — a random subset will be chosen while the other entries are discarded. Then, this subset will receive the update messages. A receiver now will forward this message to other nodes which it thinks are in the AOI of the originator. To prevent infinite and cyclic forwarding each message has a hop-limit. Further, even few missing position updates will not have such a great impact on game experience as only a jolting in game play will be noticed. Figure 2.4b shows this mechanism.

As sensor nodes are just normal nodes hosting a player instance they are subject to movement too. Therefore, the linking between a node and its sensor node is dynamic and subject to change. Every node asks its directly connected sensor nodes for better ones periodically. In case a sensor node knows of a

[2] The time to be able to reuse a connection is defined by *TIME_WAIT* and is usually four minutes.

more appropriate node it will send back this suggestion and the originally asking node will use the new node as sensor node. As mentioned earlier, sensor nodes are nodes that are just outside the AOI. The AOI is supposed to be circular around the relevant node. The sensor nodes are then outside of this circle. pSense divides this outside area into smaller sectors which is illustrated in figure 2.4a. Each sector has one sensor node[3].

Like many P2P overlays pSense also requires the joining node to be aware of at least one node in the overlay. If the known node is inside the AOI of the joining node, position updates will be sent immediately by the new node. Thus, other nodes can receive these updates because of the forwarding and may add the node to their near list. In the other case the new node will issue a sensor node request which the existing node[4] will answer with better suitable sensor nodes. However, this answer may not be the closest node possible in which case these steps are repeated until a sufficient sensor node has been found. After that, position updates of other peers in the same or overlapping AOI will be received by the new node and thus are made known by adding them into the near list.



(a) Overlay Topology Including Near (N) and Sensor Nodes (S) for one Relevant Node **(b)** Simplified Example of Message Dissemination in pSense

Figure 2.4: pSense P2P Overlay

Maintaining the sensor nodes and forwarding position update messages increases the overall traffic. Especially, with rapid game movement in mind, sensor node changes may occur often. Further, forwarded updates increase the latency. This happens particularly in hot spot areas where many peers share a similar AOI.

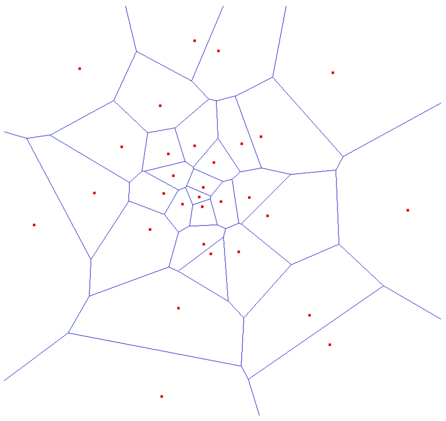
2.4.2 VON

According to Hu and Liao (2004); Hu et al. (2006), scalability is the biggest problem of nowadays NVEs which includes MMOG as a subset. Centralized architectures cannot cope with an ever increasing amount of concurrent players and, therefore, they try to tackle these scalability issues using a *Voronoi* based P2P approach called *Voronoi Overlay Network* (VON). Voronoi diagrams as illustrated in figure 2.5a were researched centuries ago. They are well studied and entail some advantages for figuring out the neighbors of a particular node. Basically, Voronoi diagrams are usually applied in two-dimensional space

[3] Note, that this applies to every node in the overlay. That is, every node has its own AOI and thus different surrounding sensor nodes.

[4] At that time, it is the only known node.

and contain a number of points — the Voronoi *sites* — as a base. On the basis of these, the plane is divided into cells referred to as Voronoi *regions* around the sites. After these regions are constructed, any arbitrary point in that region is closer to the regions' site than to any another region.


(a) Voronoi Diagram for a Given Set of Points on a Plane and the Resulting Voronoi Regions

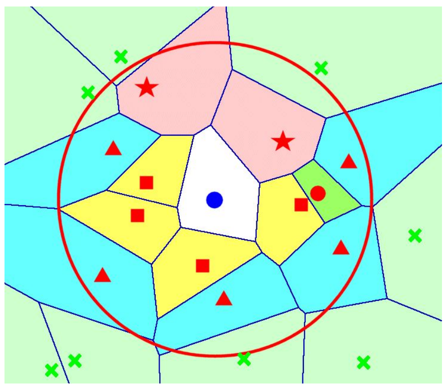

(b) Nodes in the VON Overlay (Hu et al., 2006)

Figure 2.5: VON P2P Overlay

Hu et al. (2006) use this approach to model the AOI in MMOGs which is shown in figure 2.5b. Therefore, every player constructs a Voronoi diagram based on the available information. Obviously, all players are Voronoi sites and form the regions around them. Further, a classification of neighbors is done that separates *enclosing neighbors* which share a common edge from *boundary neighbors* which are still in the AOI radius. The big advantage of Voronoi diagrams can be seen when thinking about movement. Voronoi adapts the regions according to all players positions and can thus handle neighbor relations very precisely. However, in case of few neighboring nodes, VON still keeps nodes in the neighbor list that are outside of the AOI in order to keep the Voronoi diagram valid.

Joining the overlay also requires a bootstrap node. After this, the new node sends its coordinates into the overlay using a greedy forwarding strategy. Eventually, the node closest to the new nodes position will be reached which in turn sends back a list of its enclosing neighbors. Then, the new node can build a Voronoi diagram based on the received data and notify the other nodes. Further, moving nodes send update messages to all connected nodes. Receiving boundary neighbors check if their enclosing neighbors are now visible to the moving node. In this case the moving node is notified. Leaving the network is achieved by simply disconnecting with no additional message.

Propagation of position updates between the peers is done using direct connections that all players in the AOI maintain. Neighbors leaving the AOI will get disconnected. To discover new nodes, the boundary nodes inform the player about new potential neighbors. As a result, the player maintains connections to all enclosing neighbors. Further, as the detection mechanism is done locally and does only affect a small part of the whole network, scalability seems good. To slightly minimize the problem of hot spots, VON decreases the size of the AOI after reaching a certain threshold value. After the hot spot situation stopped the AOI's size is reset.

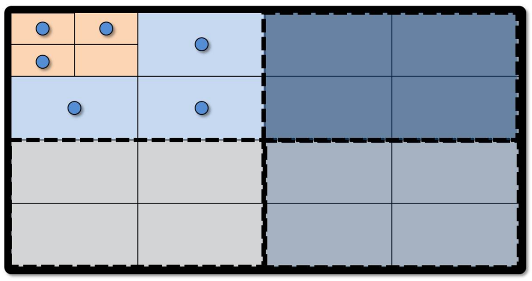
However, applying VON does not always fit the application. Especially when using an open-world scenario in three-dimensional space, VON cannot be applied easily. It may be mapped to a two-dimensional plane but at the cost of precise AOI detection in the third dimension. Further, fast players may move too fast for the boundary neighbors to send notifications. Thus, the fast moving player cannot be detected in time. As mentioned before, hot spot areas are problematic and cause the players to maintain many connections at the same time. The number of connections in the worst case would increase quadratically (Hu et al., 2008) which narrows scalability drastically. Therefore, Jiang et al. (2008) proposed a technique to build Voronoi based spanning trees: *VoroCast*. Thus, instead of directly connected peers a multicast tree is created for the Voronoi diagram. Messages are delivered using forwarding semantics. However, this increases latency even when there is abundant bandwidth.

The VON overlay can also be used to manage state information that is required in distributed games to maintain a consistent game world. This approach by Hu et al. (2008) is called *Voronoi State Management* (VSM) and builds on top of VON. Further information can be found in the cited paper.

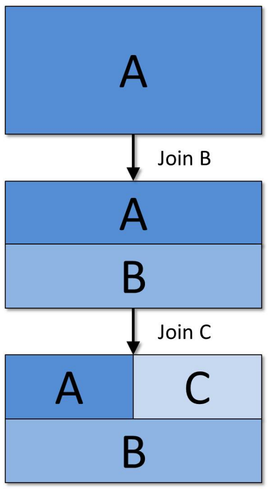
2.4.3 GP3

Esch et al. (2008) propose the *Grid-based Plane Partitioning Protocol* (GP3) P2P system. Its primary aim is to enable global-scape distributed virtual environments with a plethora of concurrent users. Further, the system supports location based queries that are of particular interest.

Technically, GP3 is the overlay protocol in the background of the *HyperVerse* infrastructure by Botev et al. (2008) which is a hybrid two-tier P2P system. Tier one features a dependable public server infrastructure. It is responsible for tracking all connected client nodes and announcing peers whose AOI intersect. The second tier consists of a loosely structured peer overlay. Both tiers utilize the *BitTorrent*[5] overlay for distributing data. For managing the public servers the structured P2P system GP3 is applied. It works by partitioning a rectangular plane into smaller pieces for new joining peers. Hence, the plane is organized bit by bit into a grid-like structure according to the GP3 algorithm. Each server (i. e. a cell in the grid) maintains connections to its neighboring cells. Further, every cell has an order which increases with further segmentation. For example, dividing a rectangle with order i results in four rectangles with order $i + 1$ whose combined extent add up to the original rectangle's size. This is illustrated in figure 2.6a. Joining nodes force one existing rectangle to be split into two and an increased order of 0.5. See figure 2.6b for an example.



(a) Grid-Like Characteristics of the GP3 with Cells and Nodes (Blue Circles)



(b) Nodes B and C Join the GP3 Overlay

Figure 2.6: GP3 P2P Overlay

Henceforth, finding neighbors is an easy task by just looking at the currently connected neighbors. Further, querying for farther away cells is also possible by use of the grid construction algorithm.

However, GP3 suffers from reliability and churn problems. As Esch et al. (2008) assume the public servers to be reliable and thus having low churn rates, this cannot be transferred to real-world applications involving volatile peers.

[5] `http://www.bittorrent.org`

Beyond, this approach is similar to the *CAN* P2P network by Ratnasamy et al. (2001) which organizes peers on a multidimensional Cartesian coordinate space instead of a two-dimensional plane. Due to this fact, Esch et al. (2008) state that GP3 can perform lookups and routing faster than CAN.

2.4.4 MOPAR

Yu and Vuong (2005) propose the *Mobile Peer-to-Peer overlay Architecture* (MOPAR) — an approach that combines elements of both structured and unstructured P2P overlays. Using this way, they claim better stability, scalability, and reliability in terms of neighbor discovery. To project the distributed game world into a virtual space, the approach of hexagonal zoning is utilized. Accordingly, the virtual world is divided into adjacent hexagon cells. Each of the hexagons further represents a region or zone in the world that might be a room, building, or area of limited view. MOPAR offers the player a continuous view that can include many adjacent zones. Although being a bit inaccurate, it provides a better approximation than using quadratic zoning. Cells in hexagonal zoning have an uniform orientation which means leaving old zones and entering new zones during movement results in the same number of hexagon cells. Each of such a cell has a unique ID that is used to distinguish between them. See figure 2.7 for more details.

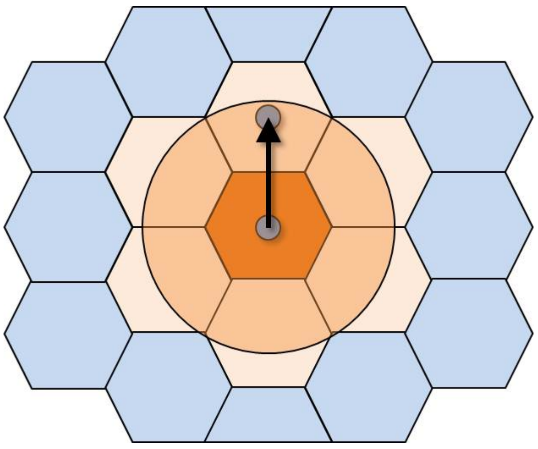


Figure 2.7: MOPAR P2P Overlay; AOI Approximation (Orange), Contained Cells (Light Orange) and Neighboring Cells (Blue)

In more detail, MOPAR uses the *Pastry* (Rowstron and Druschel, 2001) P2P overlay network which creates a DHT. Peers in Pastry maintain a leaf set with the numerical closest neighboring peers. Further, MOPAR introduces a so called *Master*-, *Slave*-, and *Home*-Node concept. Master and slave nodes are placed in the same hexagonal cell whereas each of the cells has exactly one master node and an arbitrary number of slave nodes. During movement from one cell to another master and slave nodes can change their roles. Home nodes are virtual nodes that are assigned to one cell but do not need to be located in the same cell. In contrast, home nodes do not change due to movement but in case of peers that are closer to the cell's ID than the home node before. The idea behind home nodes is to maintain the master nodes of each cell as well as master node registration. Master nodes can also query the home nodes for adjacent cells and their master nodes. This way, master nodes can exchange neighbor lists with other master nodes. Slave nodes, however, only register at the cell's master node to subscribe to information on new or changed neighbors. Slave nodes leaving the cell inform their master node about it and master nodes that leave a cell promote a capable (in terms of closeness) slave node to be the new master node.

Peers that are moving might come to the border of a cell regularly. In this case, the AOI is overlapping several cells at once. Then, the master node needs to update the slave with relevant information on the new neighbors. Additionally, master nodes utilize a movement prediction function for slave nodes in order to minimize neighbor update messages.

MOPAR seems robust as its base is the Pastry overlay. However, high churns stress Pastry as well as the MOPAR overlay. Peers that always move between two cells cause many update messages. Especially in the case of moving master nodes, a new slave nodes has to be promoted and registered at the home node. Finally, the maintenance of neighborhood lists further complicates the protocol.

2.4.5 Donnybrook

Pang et al. (2007); Bharambe et al. (2008) propose *Donnybrook*, which is a distributed system aiming to support hundreds of simultaneous players in fast-paced multiplayer games. Especially very heterogeneous bandwidth availability is a central aspect of Donnybrook. Another important issue is the efficient dissemination of updates throughout the interested peers with minimal lag.

Donnybrook's design is based on three rules: players have bounded attention, interaction must be timely and consistent, and realism should not be sacrificed for accuracy (Pang et al., 2007).

Usually, human players only have a restricted area of focus which is limited to a certain amount of actions at the same time. Thus, game objects can have different update rates dependent on how important they are, judged from a player's point of view. Therefore, Donnybrook uses *focus sets* which contain the player's attention rating for other entities. Based on this rating the update rate of position updates is adjusted. Close-by objects are rated higher as they are more likely to be in the player's center of view. Additionally, players which are aimed at are also pushed in their rating as they are of obvious importance. Finally, players that were recently interacted with also get a higher rating. All these three rating components are then weighted using dynamic factors and combined for an overall rating of the current situation.

Next, direct interactions between players should have no palpable delay and produce the same result at both sides. This includes consistent interaction between the multiple players as well as players and game objects[6].

Last, game realism should be prioritized over (little) deviation of positions. Even out-of-focus objects can get a players attention easily if they violate game logic. For that reason, Donnybrook uses *Doppelgängers* for players outside of the interest set. These are replicas of other players that apply prediction of movement and actions. Basically, Doppelgängers are computer guided bots based on the properties of real updates by the other player. In contrast to updates from players in the interest set Doppelgänger bots only receive compact updates in a lower interval. Concluding, this concept can be seen as a more advanced version of *dead reckoning* (Pantel and Wolf, 2002).

For message dissemination multicast trees are applied. Each peer is the source of a multicast group and other interested players are subscribers to this group. Donnybrook uses short-lived trees as they are subject to change often. However, these trees are only roughly optimized due to their volatile manner but still achieve satisfiable results (Bharambe et al., 2008). Further, low-bandwidth nodes are assisted by stronger nodes by using their bandwidth to propagate update messages.

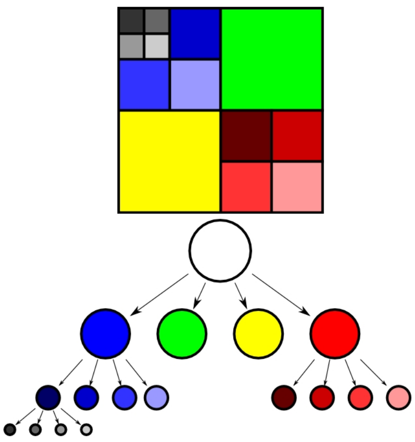
2.4.6 Quadtree-based Approach using Chord

Douglas et al. (2005); Tanin et al. (2006) present an approach for *Enabling massively multi-player online gaming applications on a P2P architecture*. Their approach is based on a quadtree and similar to the *CAN* overlay (Ratnasamy et al., 2001). A quadtree decomposes a plane space into rectangles of equal size recursively. Each of these rectangles is then assigned to a quadtree node which is equivalent to the smallest enclosing rectangle. Figure 2.8a illustrates this.

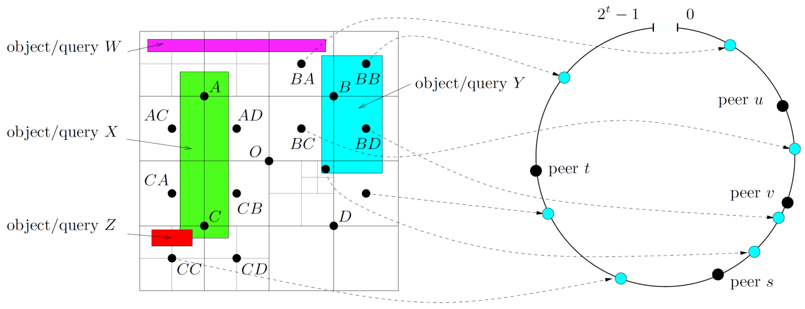
In this approach, each rectangle of the decomposition has a centroid called *control point*. This control point has certain coordinates which are stored in the structured DHT-based *Chord* overlay (Stoica et al., 2001). Thus, every rectangle is mapped to the DHT and finally to a node. As every peer knows of the

[6] E.g. "Who was first to pickup a object?" or "Who shot first?"

method the quadtree is created, everyone can build the tree and compute the coordinates of certain regions, respectively. Querying for a region in the plane space results in computation of the control point(s) which then can be queried directly using their coordinate hash in Chord's DHT. Additionally, this approach supports the storage of spatial data in the overlay. This data can in the simple case be located in a region or in the more common case span multiple regions. In this case it may be necessary to further split regions in order to project the spatial data. Tanin et al. (2006) present an algorithm in their paper which illustrates this. Further, depending on the used hash function, a good load-balancing is possible and can be done without additional effort.



(a) Quadtree[7]



(b) Quadtree, Regions, and Chord (Tanin et al., 2006)

Figure 2.8: Quadtrees

2.5 Related P2P Publish/Subscribe Systems

The following sections introduce some representatives of Pub/Sub systems which are based on the P2P paradigm. These include Colyseus, SIENA, S-VON, and Reach.

2.5.1 Mercury and Colyseus

Colyseus by Bharambe et al. (2006) — an extension of Mercury (Bharambe et al., 2002) — is a distributed Pub/Sub architecture for NVEs. Mercury is based on a range queryable DHT that allows content-based attribute matching. Its routing algorithm is similar to Chord (Stoica et al., 2001).

Mercury focuses on building a content-based Pub/Sub system for NVEs. Therefore, Mercury introduces a simple SQL-like subscription language. Subscriptions are built from conjunctions of predicates. These include data types like `float`, `string` and Boolean operators. Hence, range queries can be expressed using this language. For example, a subscription for a player in the world could be $x > 42 \land x < 300 \land y > 47 \land y < 305$ to define a player's AOI.

Mercury then defines *attribute hubs* where each hub is responsible for one attribute. A hub has a circular arrangement like Chord, hence each node is responsible for a certain area of the possible attribute space. When registering a subscription it will be sent to an arbitrary hub containing one of the present attributes. The chosen hub will then store the subscription on the corresponding nodes. When a publication is made, it will be routed to all hubs whose attributes are included in the publication. This is due to the fact that Mercury cannot predict on which hub a subscription is stored. Thus, flooding the publication to all hubs is necessary.

[7] Image by Benjamin Eikel (CC BY-SA 3.0)

While Mercury primarily focuses on content-based P2P, Colyseus changes its emphasis to object replication and consistency in NVEs in a low-latency manner. This is done by decoupling object discovery and replica synchronization, proactive replication and prefetching of relevant objects. Colyseus differentiates between mutable and immutable game objects. Of interest are only mutable objects as they are subject to change. These mutable objects are stored in Colyseus' object store. Each object can be replicated in the system for better performance and is weakly consistent. Therefore, special measures need to be taken to keep replicas synchronized with the primary copy. The primary copy only resides on one node and handles serialization of incoming updates. If a node is interested in an object from which no replica yet exists, it queries the DHT to request a replica from the primary object. This is usually the case when an object enters the AOI of a player for the first time. After that, the replica can be used by the player for interaction. Updates between the replica and the primary copy are delta-encoded to save bandwidth. However, changes to the replica are tentative. The final decision which of the probably competitive updates is chosen at the primary copy. Further, proactive replication is applied for detecting short-lived objects faster. It is based on the observation that new objects spawned near a player may be of interest for other players which are already interested in this player.

Particularly, interest prediction and prefetching of replicas are introduced. Moving objects normally follow a specific pattern. Especially slow moving objects are of special interest which can be easily predicted by their current speed. After the predicted time is calculated, the area surrounding the predicted object can be subscribed already. Finally, objects can be prefetched to decrease latency. However, prefetching increases bandwidth usage and needs to be chosen wisely.

2.5.2 SIENA

Carzaniga et al. (2000) proposed *SIENA* (Scalable Internet Event Notification Architectures), an early system that builds a content-based Pub/Sub on top of a P2P architecture. It focuses on expressiveness and thus can create comprehensive filters. SIENA's data model formally specifies all required components like filters, attributes, or patterns. A filter consists of several combined attribute constraints which are defined through a tuple consisting of data type, key name, binary operator, and value. While filters do not allow nesting, patterns can define dependencies. That is, a pattern consists of several filters in a specific order and matches only if all filters themselves are fulfilled. Further, SIENA introduces *covering relations* that define, how multiple attribute constraints for one key are handled, as there is no way to combine attributes using Boolean logic.

Among a traditional C/S approach, SIENA also includes a P2P implementation. As an event notification system SIENA is also based on the concept of brokers which are nodes that distribute subscriptions and publications in the network. Thus, the P2P architecture consists of interconnected nodes which serve connected peers and resembles superpeer overlays. Server nodes keep a list of made subscriptions including the subscribers. The server node then forwards the subscription to some neighbor server nodes. Similarly, notifications are also sent to multiple server nodes which can then check if they have connected nodes that share an interest in this.

SIENA has a flexible subscription engine but suffers from other issues. Most importantly, reliability of the overlay is low and the cost of administration is quite high (Carzaniga et al., 2003).

2.5.3 S-VON

Hu (2009); Hu et al. (2010) introduce *Spatial Publish Subscribe* (SPS) as well as the *S-VON* P2P overlay. S-VON is a general purpose overlay for NVEs which enables spatial subscription and publishing through peers. For that, Hu et al. (2010) introduce point and area publications where the publication is made at a specific coordinate or rather in a defined area. Likewise, point and area subscriptions are possible. S-VON is a content-based Pub/Sub system.

Like other P2P overlays S-VON also uses the superpeer approach (see section 1.1.2) whereas superpeers here are called *relays*. Relays are thought to be publicly known nodes with direct reachability. They are interconnected by forming a relay mesh. The task of relays is to administer the overlay and to deliver messages to peers. That is, if a peer wants to publish a message to a subscriber it will be sent to its relay which in turn forwards it to the relay responsible for the subscriber. This subscriber's relay will then send the message to the subscriber. Note, that avatars from the NVE can reside on both peers and superpeers. As superpeers are meshed, the hop-count for a published message will be at most three but may be two in case both peers are connected to the same relay. To minimize latency, peers are thought to be aware of their physical coordinates in the world. Thus, they can connect to a relay that is in close physical distance. This task of finding the right relay and getting coordinates will be done by a bootstrapping server called *gateway*.

Technically, S-VON extends the VON architecture as described in section 2.4.2. However, changes needed to be made in order to support Pub/Sub. First, each publishing VON node additionally stores the list of current subscribers to simplify message delivery. Second, Voronoi diagrams are not only kept at the nodes anymore but at the relays. Thus, a relay has diagrams for all managed peers available. This is necessary for the handling of subscriptions or publications at the relay nodes.

As S-VON is an extension to the VON overlay it suffers from the same drawbacks. Additionally, a gateway server and the concept of superpeer relays make the overlay more error-prone and complex.

2.5.4 Reach

Perng et al. (2004) propose *Reach* — a content-based Pub/Sub system that is built on top of a P2P overlay with rendezvous routing abstraction. In Reach, every event and subscription has a n-bit attribute identifier that describes which attributes this event contains. For example, setting the first bit to "1" means that this event contains the first attribute in an attribute list. Thus, matching of subscriptions against publications can be achieved by comparing both identifiers binarily. Further, a hierarchy based on the identifiers can be built where the parent identifier contains at least all attributes from its children. Every node in Reach acts as a rendezvous point for some subscriptions and messages and is responsible for forwarding messages. Subscriptions are also stored on rendezvous nodes where the later matching and forwarding is done.

A node is identified by a node identifier (with the size of m-bits) that can be less or equal to the attribute size. A mapping between node identifier and attribute identifier is done so that a node hosts attribute identifiers that share the lower m bits. Thus, a node can be responsible for multiple attribute identifiers. For routing, Reach applies a Hamming-distance scheme where each node stores a routing table containing all node identifiers which are one Hamming-distance away. Routing can then be done easily by forwarding the message to the next node being one Hamming-distance closer to the identifier. Finally, the traveled distance in hops is equivalent to the total Hamming-distance between two nodes.

Dissemination of publications is done by sending the message to the rendezvous node. It will then be forwarded to the child nodes. Each node checks if it stores a matching subscription in which case it will be forwarded to the interested node.

The dynamic characteristics in P2P overlays in Reach require identifier remapping. For example, a joining node could force an existing identifier-exhausted overlay to extend the identifier by one decimal place. Thus, routing tables need to be adjusted as well as some node identifiers. The same issues applies to correctly leaving nodes. Hence, Reach cannot handle high churn rates as the protocol expects joining and leaving in a serialized fashion. Churn, however, can occur simultaneously which results in wrong routing tables and big gaps in the Hamming-distance in the nodes. Recovery mechanisms are not addressed by Reach making it not feasible in volatile P2P networks.

2.5.5 Other Systems

Another DHT based system is *SimMud* by Knutsson et al. (2004). It is also built on top of the structured P2P system Pastry as well as Scribe (Rowstron et al., 2001) which is a topic-based Pub/Sub system with the ability to support large number of subscribers and topics. Neighbor detection is also done by Pastry's nearest neighbors list whereas SimMud mainly focuses on game state replication and conflict resolution.

An example for a system building on top of an unstructured P2P system, Rahimian et al. (2011) introduce *Vitis*. Vitis is a topic-based Pub/Sub system that is based on a gossip-based overlay and rendezvous routing. Gossiping is utilized in order to exploit subscription similarities and finally form a small-world network with nodes of similar subscription interest. Finally, publications are relayed to these clusters where the publication will be flooded to all containing nodes.

2.6 Summary

Many different systems, approaches, and implementations exist in the area of spatial P2P and Pub/Sub. However, it has to be noted that many systems apply a structured P2P approach whereas only a minority uses unstructured systems. This is expected, especially, in distributed Pub/Sub as lookups for subscriptions are easy to query. The presented systems focus on different issues in the area of NVEs. These include interest management, bandwidth management, object replication, and reliability. However, no such system can fulfil all requirements at the same time.

3 Concept

This section proposes a novel P2P-based spatial Pub/Sub approach for interest management in NVEs. After motivating the system's properties in section 3.1, a general approach is presented in section 3.2. A more practical solution is introduced in section 3.3. Later, the use of the BubbleStorm P2P overlay is explained and how it is applied in order to create a Pub/Sub mechanism.

3.1 Motivation

Distributed applications present multiple challenges that need to be coped with. Particularly, NVEs aggravate the issues by adding more constraints. In general, the challenges include:

Neighbor detection: Detecting neighbors is a crucial task in MMOGs. Neighbors are generally defined as peers that are in close proximity and reside in the AOI of a player, seen from the virtual world's perspective. In general-purpose P2P overlays neighbor detection is only rarely applied. Some approaches like Pastry keep neighbor lists to improve routing and accordingly decrease the number of hops. However, detection of neighbors in the area of MMOGs is part of normal game logic. In addition, neighbors are likely to change quickly. Bharambe et al. (2008) state, that the content of a peer's neighbor set changes with an average of 68 % per second. Thus, the neighbor detection mechanism needs to be fast and moreover reliable and fair. Reliable detection implies that neighbors can be detected with a high precision regardless of rapid movement. Fair in this context means that both neighbors should know of each other at the same time so that no player has a benefit.

Latency: Many applications that apply P2P technology can get along with a moderate or even high latency in the magnitude of few seconds. This includes distributed file storage or similar solutions where direct user interaction is not required. Again, MMOGs cannot accept high latency as the game play and user experience would suffer severely. Thus, another important goal in a multiplayer game setting is to minimize latency. However, there are certain cases where a suboptimal latency can be accepted. For example, a round based game does not require the latency to be as low as in a FPS with rapid game play.

Rapid movement: The players of MMOGs are located at a specific position in the virtual game world. During movement their positions need to be propagated to the surrounding players. Based on which approach is applied the propagation of messages can result in long paths and many hops. Especially when message forwarding is used the number of hops increases. This is aggravated by the fact that rapid movement generates many update messages and other overlay maintenance messages. All this results in an increased latency and hence affects user experience negatively.

Hot spots: Hot spots in MMOGs are areas where many players are located at the same time. Often, game play rules will encourage many players to follow or pursue one specific player. This will cause a high bandwidth demand and may easily lead to overloading the peer's capacity. As a consequence, this peer is not able to perform normal message updates which will result in unsatisfactory user experience among other things.

Connection handling: Peers need to maintain many connections at the same time due to the propagation of messages. Therefore, an efficient way of coping with this situation needs to be chosen in order to minimize the overhead of connection setup and tear down. Similar to hot spots, peers that interest many other peers often have to keep many connections open at the same time. As introduced in section 2.3, the right choice with regard to the used protocol must be made.

MMOGs with an open virtual world introduce additional peculiarities. Open worlds have a practically unlimited and continuous view. In contrast to usual games where visibility is strictly limited to rooms, buildings, or environmental structures, open worlds do not have this limitation. They may however have obstacles that hamper direct view. Further, three-dimensional virtual worlds require more precision regarding the coordinates. A simple two-dimensional mapping of three-dimensional coordinates will cause too many inaccuracies. For example, mapping a three-dimensional space to a two-dimensional counterpart will remove one axis and, obviously, detect more players as now two of three coordinates match. In addition, traditional zoning approaches are not feasible as the open world scenario is contrary to the zoning concept which basically builds on top of spatially divided regions.

To achieve a practical solution a novel spatial Pub/Sub system is developed. As opposed to most approaches introduced in section 2 this system is based on an unstructured P2P system. As traditional Pub/Sub systems distribute events without the knowledge of spatial data, this needs to be integrated as well. Therefore, area based subscriptions will be applied. Choosing the correct Pub/Sub region is crucial to the performance of the overlay. Thus, using the player's AOI will be the right solution. Choosing a bigger subscription area will result in too many received events while a small area leads to too much overhead due to overlay maintenance.

3.2 Generic Approach

Interest management is a crucial issue in NVEs. Therefore, it should be efficient by means of resource usage and fast by means of minimal delays. Figure 3.1 illustrates a generic mechanism to query for new neighbors, which follows the steps listed below given a three peer overlay network.

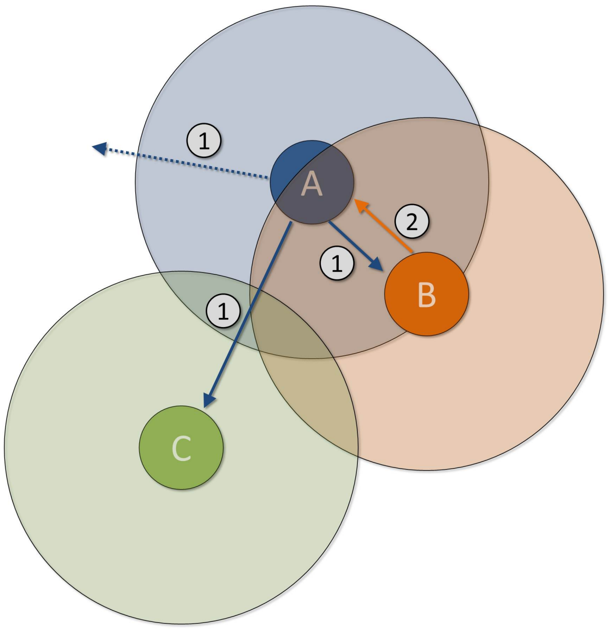


Figure 3.1: Generic Example of How to Discover Unknown Peers for a Given AOI

1. Peer *A* recently joined the overlay network and wants to know, which other avatars are in its AOI. Therefore, *A* sends a query containing its coordinates into the overlay. In a naive approach this will be achieved by flooding the network with the coordinates.

2. *B* and *C* now check if their position matches the query's AOI.
3. Assuming that only *B*'s position matches *A*'s AOI, *B* will send back its contact data.
4. After *A* receives *B*'s data the basic neighbor detection is complete.

Obviously, this basic approach cannot produce feasible results in real world environments as flooding will cause severe scalability issues. Important improvements can be made to tackle the problems. These are presented in the following section.

3.3 BubbleStorm-based Pub/Sub for NVE

This work proposes a novel approach to handle Pub/Sub using the BubbleStorm system. As the evaluation is done using a MMOG, the focus is on interest management in a three-dimensional space. The handling of many concurrent connections as well as rudimentary hot spot processing are implemented as well. However, no additional techniques are applied in this work that reduce traffic in sparse-bandwidth scenarios. The bandwidth management techniques included in BubbleStorm in combination with CUSP already reduce per-peer traffic. Section 7 will present some approaches that might help to mitigate traffic consumption but at cost of latency. Normal message propagation between nodes will be handled using directly connected peers. In a nutshell, the approach takes the following four steps to make interest management feasible:

- *Expressing interest*, i. e. subscribing to certain events in the overlay,
- *Neighbor detection*, i. e. retrieving data of peers in the AOI from the overlay,
- *Update propagation*, i. e. disseminating update messages to all interested peers, and
- *AOI monitoring*, i. e. detecting whether peers are still inside or already outside the AOI.

3.3.1 Expressing Interest

The following sections explain how BubbleStorm is leveraged to distribute subscriptions across the Pub/Sub overlay.

3.3.1.1 BubbleStorm's Bubble Concept

Section 2.2 has explained briefly the BubbleStorm system as well as the bubble concept. Usually, query and data bubbles are distinct and only share an intersection called *rendezvous node* where query and data match. Hence, results can be retrieved. This is shown in figure 3.2a. The green query bubble intersects with the blue data bubble. Placement of the data and queries on the peers is achieved by executing BubbleStorm's *BubbleCast* (refer to the following section for more details) for every data and query. The set of replicated data forms the data bubble while the set of replicated queries forms the query bubble. Data and query bubble do not need to be the same size. This becomes more clear when looking at the formulae for rendezvous probability $r = 1 - e^{-\lambda^2}$ and $e^{-\lambda^2} => e^{-dq/n}$, where dq is a fixed value for data and queries. Its ratio however can change. Choosing $dq = \lambda^2 n$ with a defined d and q, n being the number of peers and λ the certainty factor, it is clear that only the product of d and q is of interest for the same chance of matching data against queries. Table 3.1 shows some examples for varying λ and the resulting match probability.

This work makes use of the aforementioned concept in a different way. Instead of creating data and query bubbles separately they are merged. This means, that both bubbles overlap and thus create one

λ	1	2	3	4
Rendezvous probability r	63.21%	98.17%	99.99%	99.99999%

Table 3.1: Rendezvous Probability with Varying λ

congruent bubble. This is possible because the Pub/Sub overlay does not need to persist data (which are publications in this case). Publications are disseminated using other measures. Thus, only subscriptions are stored and replicated in the bubble when peers subscribe to a specific event. Figure 3.2b shows this concept.

BubbleStorm further defines several *bubble types* including instant, fading, and managed bubble types. Instant bubbles make use of the fire-and-forget semantics for non-persisted data. They are used for disseminating volatile data in the overlay. Fading bubbles can be persisted but are not maintained by peers. Thus, fading bubbles need to be updated or purged by the application periodically to keep data up-to-date. Finally, managed bubbles contain data that is owned by one peer but are maintained. The data will vanish in case the maintainer leaves the overlay.

3.3.1.2 BubbleStorm's BubbleCast for Replicating Subscriptions

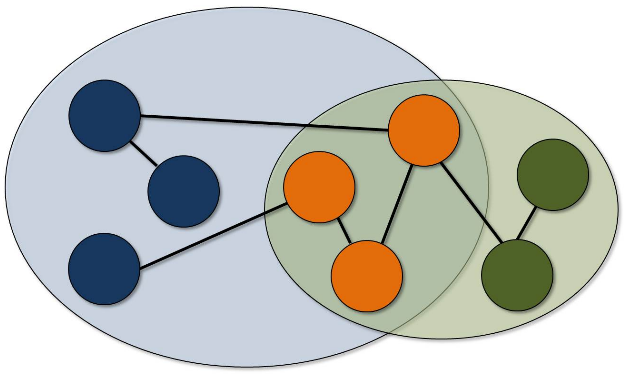
Another fundamental part is BubbleStorm's BubbleCast technique as first mentioned in section 2.2. BubbleCasts distribute data uniformly across the overlay and make sure that the given rendezvous probability is achieved. They are based on the random walk concept and combined with deterministic flooding. A BubbleCast is determined by the number of desired replicas (weight) and the split factor s. This factor defines the number of neighbors the data is forwarded to at one peer. Each receiving peer stores the data and forwards the BubbleCast to the remainder of the decreased number of replicas. When the weight reaches zero the BubbleCast stops. Thus, limited flooding with a specified number of traversed peers in a massively parallel fashion (depending on s) is achieved as well as good overlay reach. An example is illustrated in figure 3.2c with a weight of 13 and split factor of two.

The Pub/Sub approach uses BubbleCasts for distributing the subscriptions across the overlay. The advantages are a good overlay reach with deterministic probability and uniform distribution. Especially in case of high churn or peer crashes the replicas of subscriptions are likely to remain on several nodes. Further, subscriptions take advantage of the fast and parallel fashion. Publications are sent using directly connected peers for minimized latency.

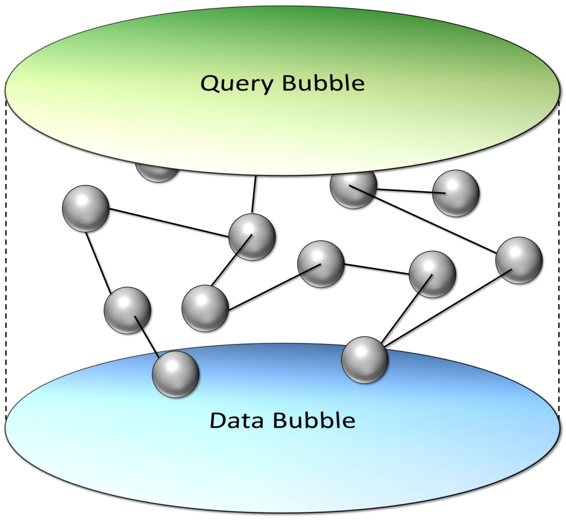
3.3.2 Neighbor Detection

Successful neighbor detection is achieved by evaluating the subscriptions. As subscriptions are sent periodically into the overlay, many peers have information available in a so called subscription cache.

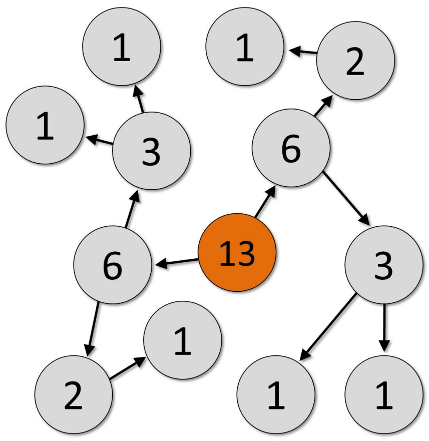
After the peers selected by the BubbleCast are asked to replicate the subscription, they will check their local subscription cache for entries that match. When this happens they will inform the subscription's originator by issuing a reply. Refer to figure 3.3 for an example. As BubbleStorm is a probabilistic overlay, no firm promise about reaching the wanted neighbors can be made. The accuracy of matching depends on several system parameters. The most important parameter is the aforementioned λ for the wanted rendezvous probability and thus the number of replicas in the overlay. Further, the update interval for sending the (updated) subscriptions can lead to different detection speeds and accuracy. However, even with a λ of only two, a sufficient reliability of over 98 % can be achieved. Thus, if a peer is missed in one round it is very likely it will be detected in the next round. Additionally, the subscription cache that includes the recent subscriptions made by other peers increases hit rates. Detection rates and speed further increase as a result. Section 5 will give some examples with varying λ and measured results. Note, that



(a) Normal Operation of Intersecting Data (Blue) and Query (Green) Bubbles



(b) Overlapping Bubbles



(c) Distributing Data to Random Peers Using BubbleCast (weight = 13, s = 2)

Figure 3.2: Expressing Interest

high churn and peer crashes do not affect the overlay in such a strong way like in structured P2P approaches. This is due to the fact that BubbleStorm requires less management overhead and handles failures well. Again, the periodically repeating subscription mechanism comes in handy to restore subscription data in the overlay after a crash.

3.3.3 Update Propagation

Publishing position updates to neighbors in the AOI timely is very important for game play and positive user experience. Miller and Crowcroft (2010) evaluated an approach using the MMOG *World of Warcraft* (WoW) where message delivery between peers was handled by a P2P Pub/Sub implementation. The results of this paper show that the latency was too high and updates were propagated too slow — on average ten times slower than direct connections. In order to achieve minimum latency between position updates and hence a smooth game play, positions are propagated directly. CUSP is used to publish position updates to directly connected neighbors. The updates are sent periodically in a high-frequency interval of about 100 ms. CUSP's streaming capability is used to achieve transmissions of

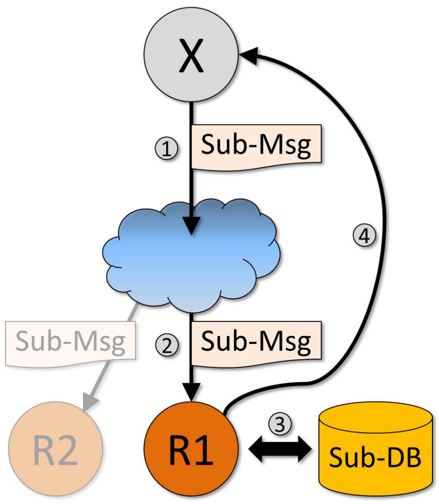


Figure 3.3: Answering Subscriptions for Neighbor Detection; Subscriber X, Receivers R1 and R2

minimal overhead and latency. Many simultaneous connections do not overload the peer in terms of connection handling. However, hot spot peers may exceed their bandwidth resources. Section 7 will present techniques to cope with this issue. However, as the focus is on MMOGs with near real-time requirements, additional overlay structures for message propagation would only increase hop count and time to deliver an update message.

3.3.4 AOI Monitoring

AOI monitoring is also required for interest management. Its task is to check if the connected peers are still in the current AOI or whether they are outside and can be removed. At first glance, this step seems trivial like the comparison of the peer's current coordinates with the last received update of another peer. However, this simple approach will create more overhead as it is very likely that the just removed peer will enter the AOI again shortly. Experiments in the scope of this work showed that this observation holds true and occurs often. From the view of a player this sudden disconnection and renewed entering in the AOI looks like a jolting movement and causes unnecessary distraction.

An improved solution is to flag peers which move outside the AOI with a special state where all operations are still executed normally. However, when a certain timeout threshold is hit, the peer will be removed. In case the peer enters the AOI again before this happens, its state is set back to normal. This helps to mitigate the problem of jolting to a large extent and decreases additional overhead.

3.4 Design and Architecture

The implemented Pub/Sub system is designed as generic as possible. In this manner, various applications can use its functionality and it is not limited to interest management alone. However, as this work's focus is on NVEs the developed prototype is tailored particularly for these systems. Figure 3.4 shows the high-level view of the architecture. The system uses a layered approach including P2P and application layer. The following sections present each layer in a conceptual way. For implementation specific details refer to section 4.

3.4.1 Peer-to-Peer Layer

This layer handles low-level configuration and is responsible for joining the BubbleStorm overlay, sending and receiving data from the BubbleStorm overlay. Note, that this layer does not distinguish between

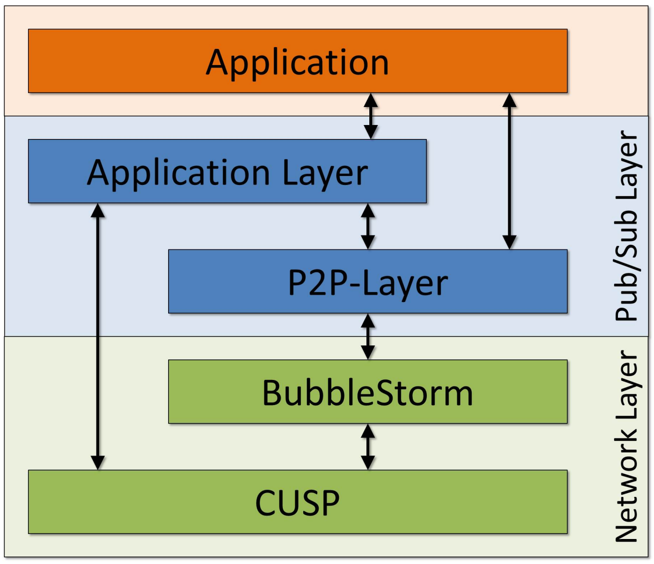
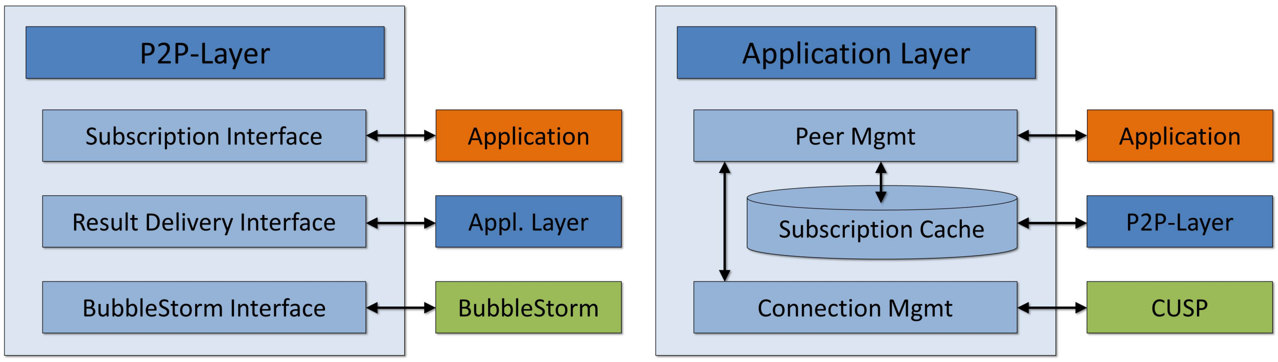


Figure 3.4: Architectural High-Level Overview



(a) Peer-to-Peer Layer **(b)** Application Layer

Figure 3.5: Detailed View of the Layers

subscriptions or any other specific form of data. It is only responsible for sending and receiving data to and from the BubbleStorm overlay and hence requires no knowledge about the structure of the data. The P2P layer solely handles subscriptions and their replies as both are transported over BubbleStorm. Other data including position updates is handles by the application layer. Exemplarily, figure 3.5a shows the P2P layer and its links to other components.

When subscriptions need to be sent, the upper layer creates a subscription message and passes it to the P2P layer. Then, the P2P layer prepares the outgoing packet for transport and finally issues a BubbleCast.

Following, other peers are chosen by the BubbleCast and receive the subscription data. They will then execute an application callback and pass the received data. Thus, the application layer can update its local subscription cache and check if there are any peers the originator could be interested in[1]. If that is the case the answer containing a list of interesting peers will be sent back directly.

These incoming replies are also routed through the P2P layer. They are forwarded directly to the application once they are received completely.

[1] Note, that it is possible that the querying peer receives its own request. However, this will only happen in networks with only a few peers and is caused by BubbleStorms initial self-connections.

3.4.2 Application Layer

The application layer lies on top of the P2P layer and is responsible for functionality that is more close to the real application level. The application layer is in charge of maintaining the subscription cache, connection management, peer management, and disseminating messages to connected peers. This layer is, in contrast to the P2P layer, already partly application specific. This means, that data structures and objects are mapped to the enclosing application. The layer is implemented this way, to achieve better performance. For example, data does not need to be wrapped to a more generic structure and then re-wrapped between the different layers.

Note, that this layer handles all data that needs to be sent or received between peers. This includes messages relevant for Planet π4 like player interaction and game state updates. Consequently, one established connection to a peer is multiplexed for both interest management and game logic updates which saves unnecessary network overhead. Figure 3.5b shows the application layer in more detail.

3.4.2.1 Subscription Cache

The task of the subscription cache is to locally store subscriptions made by other peers. As mentioned earlier, for better performance and reliability, each peer manages such a cache. Each entry stands for one peer-specific subscription and is bound to a specified timeout threshold. Subscriptions can be received multiple times from one or more peers in which case they are renewed. When the timeout threshold is hit they are purged. As subscriptions are stored completely in the cache, it can be queried to get all known subscriptions whose AOI matches certain coordinates.

3.4.2.2 Connection Management

The connection manager handles all incoming and outgoing connections. Its tasks include opening of connections to currently not connected neighboring peers and closing connections which became obsolete. That is, connections that did not transfer data for a defined time threshold or peers which are too far away and outside the AOI, get removed. Also, incoming data is received and after completion of a message it is passed to the application. Incoming data contains positional updates that get published by other peers as well as additional game or maintenance messages. Finally, outgoing messages from the application are sent to the designated target peer(s).

3.4.2.3 Peer Management

The peer management component is the central instance that configures and connects all components. Hence, the peer management component administrates all open connections and manages their states (i. e. normal, soft-remove, and hard-remove). In addition, the peer management component offers an interface for the application for message broadcasting and connection related tasks.

The abovementioned states of a connection express the AOI monitoring states. Soft-remove is the state when a player left the AOI but is still considered active in sense of update handling. After a timeout the peer will change to the hard-remove state. Consequently, the connection will be shut down gracefully in the next cycle. When a peer returns to the AOI again before the threshold is reached its state is changed back to normal.

3.4.3 Application

The application stands for the NVE software that uses the Pub/Sub overlay. Dependent on the application this part can be a wrapper component or can simply be integrated in the existing network model of the

NVE. To be more precise, this component will usually maintain an instance of the Pub/Sub component and configure it with the application specific parameters. NVEs as well as other three-dimensional interactive applications are usually constructed using a game loop pattern. That is, every iteration includes processing of user input, process network data, calculate AI, move entities, draw graphics, and so on. The Pub/Sub layer can thus be integrated in this game loop as it is required to run periodically. No separate loop is required. More implementation specific information can be found in section 4.

3.5 Summary

This section has presented the conceptual design of the Pub/Sub approach developed in this thesis. Using the BubbleStorm overlay and utilizing the advantages of its architecture, a Pub/Sub system for NVE was be implemented. This includes the approach for uniformly broadcasting subscriptions through the overlay using BubbleCasts and maintaining them. Reliability in terms of network failures and low latency in terms of message propagation can be achieved by performing regular updates of the subscriptions over many nodes and using directly connected peers.

Concluding, a Pub/Sub overlay with fixed area subscriptions and point publications was achieved.

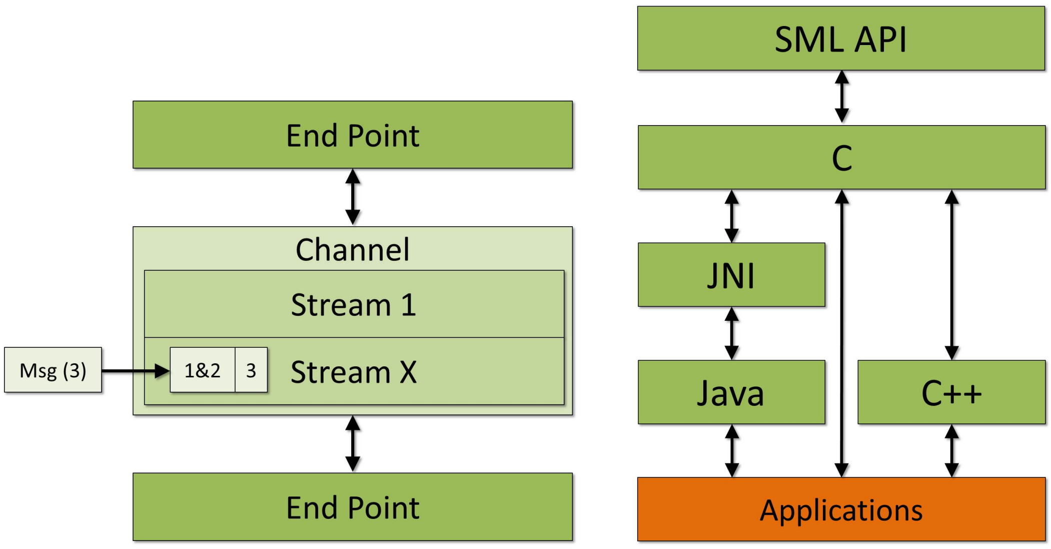
4 Implementation

This section explains the concept presented in section 3 in more detail. First, the software environment is described in section 4.1 before the developed prototypes are presented in section 4.2. This includes the Planet π4 prototype that enables Pub/Sub to be the underlying network model. Subsequently, the fine-grained architecture of this implementation is presented in section 4.3. Finally, information about the fundamental network data structures is given in section 4.4.

4.1 Environment

The following sections present the environment which this thesis' implementation is bound to. Generally, the implementation is hardware and software independent. However, compilers for the *Standard ML* programming language as well as C and C++ must be available. The implementation itself is cross-platform compatible and has been tested under GNU/Linux and Microsoft Windows.

Additionally, as explained in section 2.1 the MMOG Planet π4 is used as a basis. Planet π4 requires the *Irrlicht*[1] engine. Irrlicht is an open-source and fully featured 3D engine aiming at portability and small total size.



(a) CUSP Overview Showing End Points, Channel, Streams, and Message Fragmentation **(b)** Different Language Bindings and Interaction

Figure 4.1: Environment Details

4.1.1 CUSP's Connection Handling

Section 2.3 has introduced CUSP briefly. For a better understanding of the following sections CUSP's programming model will be explained further. Figure 4.1a shows an architectural overview of CUSP. Fundamentally, every CUSP connection operates on an *end point*. It offers traditional functionality which

[1] `http://irrlicht.sourceforge.net/`

includes connecting to a peer and listening for incoming connections. More precisely, the latter is referred to as advertising a service. However, service advertisement goes beyond simple listening as an end point's associated *channel* can have multiplexed and independent streams. Streams are distinguished between *OutStreams* and *InStreams* and operate independently. Further, each stream has an identifying service ID. A connection can be uni- or bidirectional depending on whether only one stream or two streams are established for the particular service ID. CUSP's lightweight connection handling even enables opening a stream, sending data over it, and closing the stream at the same time. Another aspect is, that data on the stream can be fragmented into multiple chunks to achieve better utilization and minimize the number of packets to transfer. The application, however, needs to cope with this peculiarity and has to combine the chunks to a complete message again.

CUSP offers an asynchronous and non-blocking network model. This means, that a call to a CUSP network method immediately returns regardless whether the operation already finished. As a consequence, CUSP requires the application to register for callback events which inform about the result of an operation.

4.1.2 Standard ML and MLton

BubbleStorm and CUSP use the Standard ML (SML) programming language specified by Milner et al. (1997). SML is a functional language originally used in the area of programming language research and theorem solving. MLton[2] is one of the major implementations and compilers for SML. MLton makes use of whole-program optimization in order to create highly-optimized and fast executables. Further, MLton can produce wrappers for C-code so that the functionality of a program written SML can be used easily with different programing languages assuming a custom wrapper exists.

4.1.3 Bindings for Other Languages

The MLton-generated C-code can be used as a basis for the development of wrappers for other languages. At the moment, BubbleStorm and CUSP are available in the C, C++, and Java programming languages. As mentioned, C-code is built-in whereas for the other languages custom wrappers need to be developed. The C++ wrapper can be developed without much effort as C and C++ share a common base and only object oriented programming has to be added[3].

Figure 4.1b shows how the wrappers work. Obviously, all of them build on top of the C library and execute the library C-functions directly. C++ only adds object-oriented principles whereas Java requires additional effort. Because Java is a memory managed language, measures need to be taken to adapt the unmanaged C-code to this. *Java Native Interface* (JNI) is used to achieve inter-language communication.

A part of this thesis has been the development of the initial bindings for the BubbleStorm API in C++. This includes the `BubbleType`, `ID`, `Measurement`, `Query`, `QueryRespond`, and `Topology` classes. However, due to several changes in the BubbleStorm API, some classes are now considered obsolete and got replaced by others.

4.2 Evolutionary Prototyping

Two milestones of a prototype have been developed during the progress of this thesis in order to decide about the feasibility. This includes a proof-of-concept prototype and an evolved version with a more elaborate implementation for the Planet $\pi 4$ game.

[2] `http://www.mlton.org`

[3] Of course, C-code can be used directly in the C++ language but at the cost of losing object-orientation and its benefits.

The prototype for a proof-of-concept evaluation was developed first. This prototype was rather simple but included distributed position updates using the Pub/Sub approach. In contrast to the final implementation of this work, the prototype missed almost all optimization techniques. For example, update messages were transmitted in a human readable textual representation for easier debugging instead of byte-efficient binary data. Further, the movement model of the peers was rather unrealistic and included only random point hopping in two-dimensional space.

But even without optimizations, this prototype was deemed promising after first tests. Hence, the concept was extended and optimized. Note, that this prototype used an older development version of BubbleStorm with a different concept of query replying.

Also, the development of the aforementioned C++ wrappers took place in this phase. In comparison to Java, the C++ bindings impose a little programming overhead. For example, the missing feature of anonymous classes in C++ requires more program code as separate callback classes need to be defined. Hence, the implementing classes have to use many of these callback interfaces which can become unhandy.

The integration of the Pub/Sub overlay with Planet $\pi 4$ was another main part of this thesis. Planet $\pi 4$, as a research MMOG, already had a pluggable network system where different network overlays can be integrated easily. Therefore, a specific API was defined that needs to be implemented by the new network overlay. The following sections will provide more information about the implementation.

4.3 Components of the Developed Pub/Sub Overlay

The following sections present a detailed view of the Pub/Sub overlay. Particularly, existing links between the layered approach are explained.

4.3.1 Peer-to-Peer Layer

This layer contains the functionality to interact with the BubbleStorm P2P overlay. It is responsible for distributing subscriptions and receiving their results. Keep in mind, that these results are not position updates but rather lists of interesting peers. Figure 4.2 shows the layer and existing links in more detail.

P2PLayer This is the base class that handles input and output from and into the BubbleStorm overlay. The methods *init()* and *search()* are accessible from the application level.

init() initializes the whole Pub/Sub solution including the BubbleStorm overlay. Also, during the configuration phase must be distinguished between bootstrapping and normal nodes for proper operation. Further, a CUSP end point is created that can be used for all following data transfers. Next, the creation of fading bubbles with the corresponding λ value is executed and the layer is registered at the notification interface. This interface is used for sending data back to the originator in a convenient way.

search() is the central method for sending subscriptions into the bubble and thus expressing interest. Sending a subscription message results in an issued BubbleCast. The subscription's data is passed directly as binary raw data to the BubbleCast.

onBubble() is the method that will be executed when a peer receives a BubbleCast and, hence, new subscription data. Therefore, the subscription data is extracted. Subsequently, a callback is executed in order to pass the received subscription to the application. This is done using the *QueryMatcher* interface described below. After the callback has been executed and results have been found, they are sent back to the subscription's originator directly. This is achieved by utilizing the *sendImmediate* method of the notification interface.

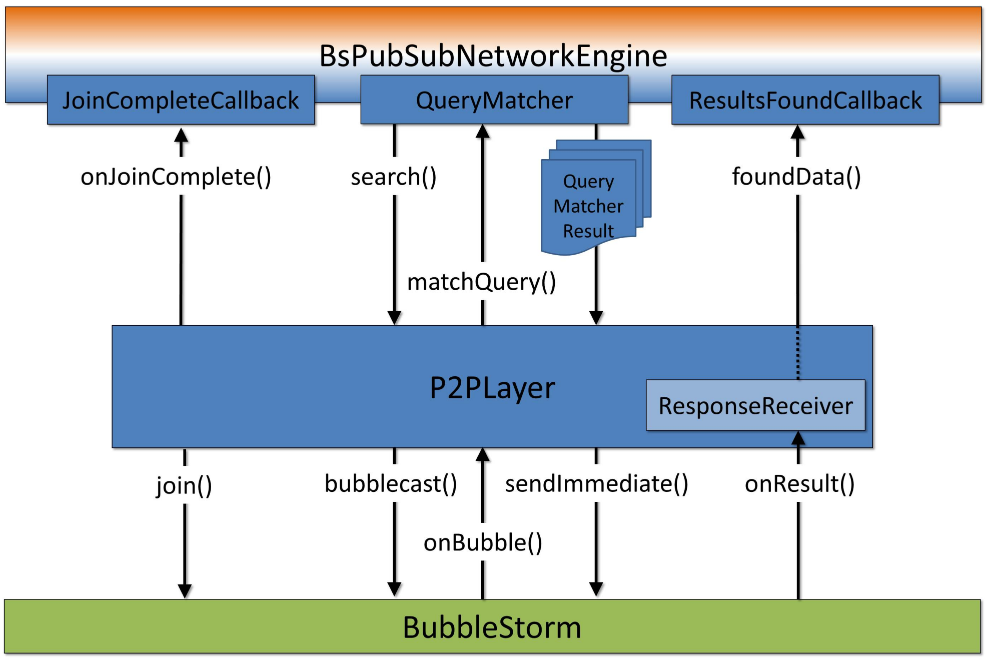


Figure 4.2: Detailed Architecture of the P2P Layer

> ***onResult()*** will be called when there is incoming data triggered by the notification interface. This data contains peers that are in the AOI and are presumably interesting for the peer. A *ResponseReceiver* instance is then created that will read all data and finally execute the *ResultsFoundCallback*. This additional step is necessary as incoming data may be spread over several packets and needs to be reassembled.

QueryMatcher The QueryMatcher interface needs to be implemented at the application level (refer to figure 4.2) and is responsible for evaluating newly incoming subscriptions.

> ***matchQuery()*** is executed by the aforementioned *onBubble()* method. The purpose is to let the application know about the new subscription and to check if there are matching results for the subscription's AOI. In case there are results for the originator, a filled *QueryMatcherResult* object is returned.

QueryMatcherResult An object of this type stores peers that are of interest for the subscription's originator. The data is again stored in a raw binary format as the overlay does not need to know about the actual data.

> ***getData()*** returns the available peers that reside in the subscription's AOI. In case there are no known peers an empty result is returned.

ResponseReceiver is an utility class for reassembling possibly fragmented incoming data packets. These packets contain replies to subscriptions with interesting peers for the originator.

onReceive() stores all received data into an internal buffer. After all data is received from the network stream the packet has been reassembled, consequently. The end of the stream is detected by a stream shutdown which triggers a CUSP callback. This, in turn, results in executing the *ResultsFoundCallback* at application level.

ResultsFoundCallback Again, this callback needs to be implemented at application level and enables the receiving of interesting peers that match the subscription.

foundData() transports a list of peers which was sent by receivers of BubbleCasts and are thus in the peer's AOI. This list can contain multiple peer entries and includes connection data required for initiating contact. Again, this method only passes raw data that needs to be interpreted by the application.

JoinCompleteCallback This application implemented callback informs that initializing and joining was done successfully.

onJoinComplete() is executed when the overlay was initialized successfully and the peer joined the BubbleStorm overlay. Normal operation can start after this method was triggered.

4.3.2 Application Layer

The application layer contains the following components that enable networking, spatial updating (i.e. publishing), and peer management. Figure 4.3 illustrates all important components and their connections.

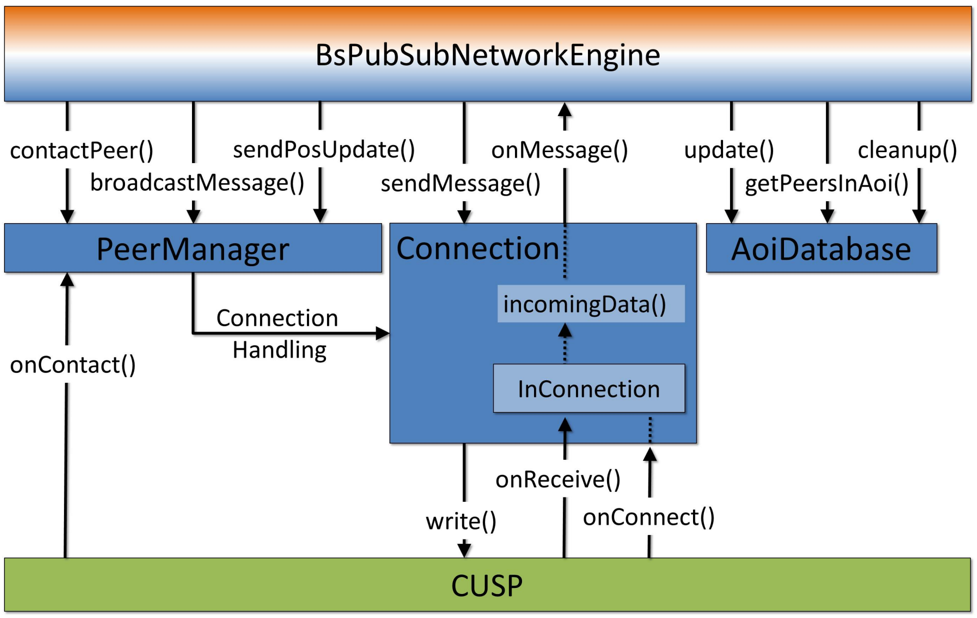


Figure 4.3: Detailed Architecture of the Application Layer

AoiDatabase This component manages all received subscriptions from other peers in the overlay and is referred to as the subscription cache. Management includes storing new subscriptions as well as updating and purging obsolete ones. In a nutshell, incoming subscriptions are matched against locally cached subscriptions from other peers.

update() receives a single subscription and stores it within the local subscription cache. The received subscription can either be already present in the cache or a new entry. In both cases, the cache has to replace an updated subscription or store the new subscription in order to be up-to-date.

getPeersInAoi() is a central method in interest management and builds a list of peers that reside in the originator's AOI. The approach is straightforward and includes iterating through all known cached subscriptions and comparison with the wanted coordinates. When the last received coordinates and the originator's AOI match the entry is pushed into a list. Finally, the list is returned in order to be sent back.

cleanup() tidies up the subscription cache and removes obsolete subscriptions. Obsolete subscriptions are detected using their current age since the last update and a timeout threshold. Note, that the entries of this cache are subject to change often. Therefore, removing obsolete entries and keeping subscriptions up-to-date is of great importance.

Connection The task of the connection class is to handle a CUSP connection to a specific peer. This includes connection setup and tear down as well as sending and receiving data from and to the connected peer. In addition, connections are monitored for failures and staleness. In gaming terms, a connection is bound to one player and hence implements Planet π4's *IPlayer* interface. This contains methods to set and get spatial positions or access other player relevant properties.

onConnect() is executed when another peer is creating an incoming data stream. This stream is given to the *InConnection* class which reassembles possibly fragmented incoming data.

incomingData() receives data from another peer. Due to CUSP's unidirectional stream concept, incoming data can open a new one-sided connection. If so, an outgoing stream will be opened for further use. Otherwise, the received message is passed to the network engine (application). Note, that incoming data is first buffered using the *InConnection*.

sendMessage() is a straightforward method for sending raw data through the stream to the connected peer.

(un)markForSoftRemoval() sets or unsets a flag that marks the connection for soft removal. The connection will stay active until a timeout value is hit and then change to the hard removal state.

(un)markForHardRemoval() sets or unsets a flag that marks the connection for hard removal. The connection will thus be removed in the next cleanup cycle if there is still no activity.

InConnection This class is encapsulated within the Connection class and exists solely for receiving data. It uses a buffering technique for incoming data as messages are subject to fragmentation. It is similar to the ResponseReceiver class in the P2P layer. The difference to this class is, that the completion of a message is not terminated with a stream shutdown. Here, an arbitrary number of messages is transferred. This class already performs simple message conversion based on raw data.

onReceive() progressively receives data and appends it to an internal buffer. Then, the buffer's content is checked for a completely received message. In this case, the message is passed to the application and removed from the buffer. Conversely, waiting for additional data is necessary. Further, primitive but fast error checking is done by comparing the received message length determined by the header value and the actual message length.

onShutdown() / onReset() will terminate the *InConnection* as the connected peer did shut down the stream.

PeerManager is the central component in the architecture. It configures and connects all components of the layer. This includes management of peers, connections, and functionality for sending and receiving data.

sendPositionUpdate() publishes the player's new coordinates to connected peers. Note, that this method is not used for high-frequency spatial updates. These are handled by the game application and disseminated using the *broadcastMessage()* method. The publications made by this method are used for the Pub/Sub overlay to handle peers that left the AOI. To be exact, these low-frequency updates are used for soft and hard removal states as described earlier. Technically, the publication message is constructed with the player's current coordinates. To disseminate the data to all connected peers the *broadcastMessage()* method is executed. Also, the *cleanup()* method of the *AoiDatabase* is triggered to remove obsolete subscriptions.

broadcastMessage() sends raw input data to all currently connected peers. The *sendMessage()* method of the *Connection* class is used for each peer.

contactPeer() performs the important task of establishing connections to peers that were received as a result of an issued subscription. Multiple peers can know the same peer *X* which is in the AOI of the subscription's originator. Due to this fact the originator may receive peer *X*'s data several times. Because of the asynchronous nature of the overlay, contacting the same interesting peer multiple times should be prevented. Therefore, an address cache is used that stores all outgoing connections attempts. If a newly received peer cannot be found in this cache, it will be added and a connection will be established (using CUSP's *contact* function on the end point). In addition, peers stored in the address cache also time out in case they are not reachable anymore.

onContact() is executed after a connection attempt by *contactPeer()* was successfully made. Again, it must be checked if the connection is already open. This can happen if the other peer also tried to contact the originator and succeeded earlier. In case of an already established connection the duplicate connection will be terminated. Further, a *Connection* instance is created that handles all incoming and outgoing data from now on. Finally, as only an outgoing stream is opened, there needs to be established an incoming stream. This is achieved by listening for an incoming attempt. Next, the peer's service ID is sent to the other peer too, so it is able to initiate contact.

onContactFail() is executed in case a connection could not be established.

onConnect() is triggered by incoming connection requests by other peers. First, possibly existing soft or hard removal flags are reset. Then, a *Connection* instance is created similar to *onContact()*.

checkConnections() performs connection maintenance and marks connections for hard removal. This is achieved by checking the soft removal flag and a defined timeout for each connection. When the threshold is hit the connection will change the state to hard removal. Also, connections that did not receive data in a certain time span are tagged for hard removal.

purgeConnections() finally removes obsolete connections by iterating over all connections with the hard removal flag set. To remove a single connection the *removeConnection()* method is performed.

removeConnection() purges a connection from the set of active connections. This additional step is required in order to keep the current set of connections in a valid state. Removing entries at one iteration could cause integrity problems. Additionally, this method is executed by other parts of the layer in case of detected problems with a specific connection. For example, it is performed after receiving invalid or bogus data.

4.3.3 Network Engine

The network engine component is the link between Planet π4 and the Pub/Sub overlay. Thus, it obeys game architecture and implements game interfaces and obviously also implements interfaces required by the Pub/Sub layer.

BsPubSubNetworkEngine This component connects Planet π4 with the Pub/Sub engine and enables interest management from the game's point of view. In the following enumeration underlined methods are callbacks executed by the Pub/Sub layer.

init() creates a CUSP end point and advertises it. Next, it initializes the main components including *P2PLayer* and *PeerManager*. Further, the low-frequency creation of publications is registered at the task scheduler engine.

connect() is the initial method that needs to be executed. It initializes the P2P and application layer and triggers connecting to the network.

onJoinComplete() tells Planet π4 that the initialization of the Pub/Sub overlay was done successfully. After this method is called, networking code can be executed.

castMessage() can be called to disseminate a message to all connected peers. This is usually the case for spatial publications.

disseminate() is a helper method that takes raw data and adds a message header before sending to all connected players. For message propagation *castMessage()* is used.

sendMessage() is used to send data directly to the specified player or object. This is usually used by the game to deliver interaction messages like collision detection or object interaction.

sendToPlayer() is similar to sendMessage but adds also a message header to raw data.

foundData() is a callback method for found results from a preceding subscription. As the P2P layer only handles raw data it must first be converted into the application's structure. After basic error handling the message's content is extracted. As a reply can contain multiple entries they need to be traversed. Each of these entries contains the other peer's coordinates and connection data. If a peer is still of interest a connection attempt will be made using the *PeerManager*'s *contactPeer()* method. Otherwise the entry will be discarded.

onMessage() acts like an interceptor. Every received message gets passed through this method before being delivered to the game application. This is necessary in order to handle proximity of peers and their states. Therefore, this method checks if the sending peer is still in the AOI, already outside, or moved back inside. In the second case it is marked for soft removal and in the last case all flags are reset. Finally, all other messages are passed to the application.

matchQuery() evaluates an incoming subscription. First, the received data is parsed. The message is subsequently passed to the *AoiDatabase* which updates its entries and later is queried for results matching the originator's AOI. In case of actual results a reply message is built and returned.

createSearchPositionMessage() creates a subscription including the current position and connection data. This message is returned and used by the low-frequency update mechanism.

executeTask() is a periodically repeating mechanism with the low-frequency update rate. It performs creation and sending of subscriptions to the overlay. Further, peer maintenance is triggered.

4.3.4 Statistics Collection

To gather more run time and performance information statistics are collected on-the-fly. BubbleStorm's statistics collector is used. This collector stores all statistics in a *SQLite*[4] database. For the Pub/Sub overlay the statistics are collected every low-frequency cycle. See figure 4.4 for an overview. Metrics include number of (correctly) connected neighbors, bandwidth usage, and so on. Section 5 will present the evaluation gained from the collected statistics under the influence of varying configuration parameters.

It is important to note that many metrics require a global knowledge of the whole overlay and all peers. Such a knowledge without a central coordinator cannot exist in pure P2P systems. Therefore, these metrics can only be collected in simulated runs. The simulation enables a global view over all participants. Hence, Planet π4 can be started in a real-time mode or in a simulation mode. This simulation mode is based on the BubbleStorm simulator engine and is described further in section 5.1.

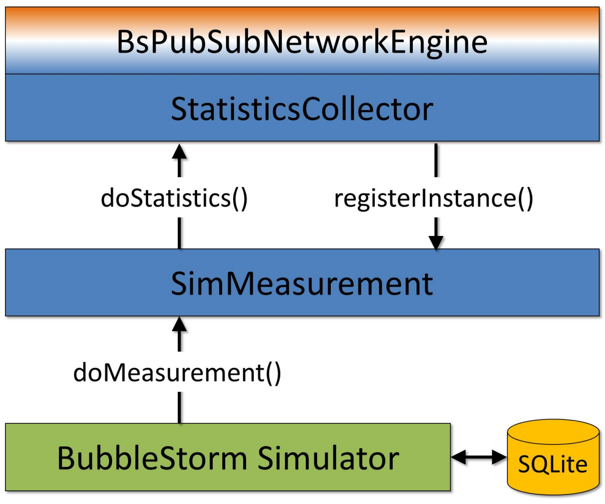


Figure 4.4: Statistics Collection and Compiling

[4] http://www.sqlite.org

SimMeasurement This class is the basis for collecting and compiling statistics. Thus, each player instance needs to register at this class to announce that it wants to provide data.

> ***registerInstance()*** stores the given statistics collector and initializes it. A statistics collector is bound to one simulated player instance. Thus, many of these collectors exist in simulations with dozens of players. Further, the interval of statistics collection is configured once.
>
> ***doMeasurement()*** is the main algorithm for compiling statistics. It works in three steps. First, the correct spatial coordinates of all registered players are stored in a list for further use. This is needed by every peer in order to calculate some metrics like precision. Next, the list of registered players is traversed and the peer-specific statistics compilation is triggered. Further, every peers receives a copy of all players and their correct coordinates. Last, global statistics are compiled. At the time of writing this thesis, only the spatial difference between the real and perceived coordinates is calculated here.
>
> ***callbackHandler()*** is executed periodically by an external method in the simulation framework. It triggers the *doMeasurement* method to compile this cycles statistics.

StatisticsCollector is the peer statistics compiler. Every collector is bound to exactly one peer and, thus, manages metrics that can be obtained by every peer without total global knowledge. However, it is possible to pass required global data into the collector by *SimMeasurement* if needed.

> ***createStatisticsObjects()*** initializes the per-peer statistics objects. This is necessary to obtain both statistics per peer and averaged statistics over all peers. The simulator model supports this automatically after configuration.
>
> ***doStatistics()*** calculates all defined metrics. They include precision, recall, accuracy, specifity, known neighbors, anticipated neighbors, and traffic. After the metrics are calculated they are stored in the simulator's database. This is done by executing a function of the simulation engine. See figure 4.4 for more details.

4.4 Network Data Structures

Several data structures are implemented for message transport over the network. Basically, every sent message contains a *MessageHeader* with information about the message type, the total message size, the issuing player, and the team. This is illustrated in table 4.1.

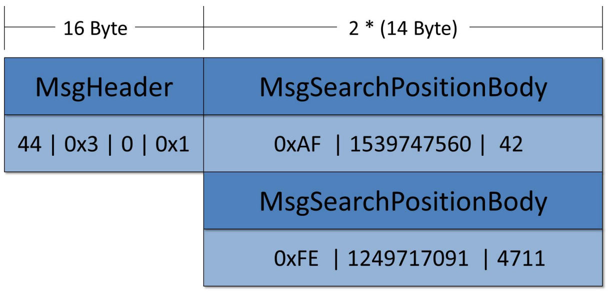


Figure 4.5: Packing a Subscription Reply Message

In addition, several message body types are defined. *MsgSimplePositionBody* is the base type and includes fundamental data required for spatial subscriptions. All fields are listed in table 4.2. The field *visionRadius* is also included for the case that different game entities can have varying field of view and thus a variable AOI. The *timestamp* field enables the message receiver to check if the data contained in the message body is still of interest or already stale. This simple spatial message is constructed when subscriptions are made.

Bytes	2	2	4	8
Fields	msgSize	msgType	team	playerId

Table 4.1: MessageHeader Data Structure

Bytes	4	4	4	4	4
Fields	posX	posY	posZ	visionRadius	timestamp

Table 4.2: MsgSimplePositionBody Data Structure

Next, *MsgSearchPositionBody* is the message type that is used for answering a subscription with a list of fitting peers. As multiple peers can be of interest, the replied message can contain several items each structured through a *MsgSearchPositionBody*. Table 4.3 contains the fields used for each item. They include contact information with IP and port as well as the player's id for identification reasons. An example of packing a subscription reply together with header and body is shown in figure 4.5.

Bytes	8	4	2
Fields	elementPlayerId	ip	port

Table 4.3: MsgSearchPositionBody Data Structure

Last, it is important to note, that the publication's message structure is not part of the Pub/Sub overlay. They are defined by Planet π4 and are independent of the overlay. For the sake of completeness, these messages contain the coordinates and rotation of the player. Further, they include the velocity of both position and rotation which is required for dead reckoning (Pantel and Wolf, 2002).

4.5 Summary

This section has presented the architecture of the Pub/Sub overlay implemented in this thesis. Development included fundamental work on the C++ bindings and, naturally, the Pub/Sub overlay. In addition, Planet π4 was enhanced by implementing a global settings manager. This allows a simplified configuration of all relevant parameters especially network and game properties. Further, it simplifies simulation as settings profiles can be passed and used by the instances.

From a programmer's perspective the design and API was kept simple and structured. However, due to many dependencies from BubbleStorm, CUSP, and Planet π4 some programming overhead was unavoidable. Concluding, the two-layered approach is still easy to implement from the application's point of view.

As Planet π4 and BubbleStorm are under active development and thus are subject to change, further improvements based on this thesis are likely to be implemented.

5 Evaluation

This section evaluates the developed Pub/Sub system and shows how it performs under load. First, section 5.1 will explain the simulator in more detail and show how it works. The metrics evaluated in this thesis are presented in section 5.2. Section 5.3 will introduce the evaluation environment and setup and the achieved results in section 5.4.

5.1 Simulator

The simulator was briefly introduced in section 4.3. It is a simulation framework which is part of the BubbleStorm project and written in SML. However, basic C++ bindings exist. The simulation framework provides an extensive feature set for different network parameters. For example, packet loss, jitter, and typical delay between two geographically positioned end points can be configured to simulate real-world networks. For this thesis however, the focus is on more application-specific metrics.

The simulator uses a discrete event-based simulation model. This means, that two simulations with the same parameters will produce the exactly same results. Further, all operations are triggered by events that are executed in a chronological sequence.

Figure 5.1 shows how simulation results are collected and stored. Each application instance is bound to a statistics collector. These perform the operations necessary for per-peer statistics and enable global statistics. As mentioned before, the simulator framework stores all collected data in a SQLite database for later result compilation.

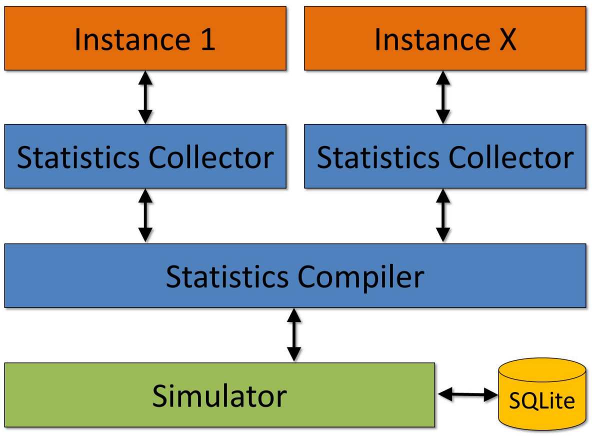


Figure 5.1: Overview of Simulation

5.2 Defined Metrics for Evaluation and Comparison

The following metrics are applied to measure quality of the developed Pub/Sub overlay.

5.2.1 Metrics Deduced from Information Retrieval

The computer science area of information retrieval defines important metrics which are also of interest for this thesis. Manning et al. (2009) state *precision* and *recall* as the most important representatives for measuring the effectiveness of a system. These first two metrics build the foundation in combination with further information retrieval metrics like *accuracy* and *specifity*. Table 5.1 defines the terms true and false positives as well as true and false negatives. In the following, the are referred to as *TP*, *FP*, *TN*, and *FN*.

For example, consider quality checks for new cars at the end of production where each car can either have the result flawless or faulty. Thus, TP are faulty cars correctly checked as faulty (hit), FP are flawless cars incorrectly checked as faulty (false alarm), TN are flawless cars correctly checked as flawless, and FN are faulty cars incorrectly checked as flawless (miss).

	Relevant	**Non-relevant**
Retrieved	true positives (TP)	false positives (FP)
Not Retrieved	false negatives (FN)	true negatives (TN)

Table 5.1: Contingency Table

Precision P is the fraction of retrieved documents which are relevant for the query (Manning et al., 2009). Precision P is defined by

$$P = \frac{\#\text{relevant items retrieved}}{\#\text{retrieved items}} = \frac{TP}{TP + FP}$$

Recall R is the fraction of relevant documents which are retrieved and all relevant documents. Recall R is defined by

$$R = \frac{\#\text{relevant items retrieved}}{\#\text{relevant items}} = \frac{TP}{TP + FN}$$

Precision and recall have to be evaluated together. For example, a perfect recall of 1 can be achieved easily by retrieving all documents for a query ($R = x/(x + 0) = 1$). However, precision in this case will be very low due to plenty false positives.

Another derived metric is accuracy A which is the success ratio of a queries result. It is defined by

$$A = \frac{\#\text{correct items}}{\#\text{all items}} = \frac{TP + TN}{TP + FP + TN + FN}$$

Finally, specifity S is the fraction of correctly classified negative items in relation to all negative items. It is defined by

$$S = \frac{\#\text{correct negative items}}{\#\text{all negative items}} = \frac{TN}{FP + TN}$$

In a NVEs scenario with the focus on interest management, precision stands for the ratio of peers that match the peer's AOI in relation to all received peers. Recall is the ratio of matching peers in relation to all existing and matching peers. Thus, the neighbor detection should have a high recall so that all neighbors in the AOI are found. However, precision should also be high so that only correctly peers in the AOI are retrieved. Low recall and precision will lead to a bad detection rate and affect game play negatively. Further, accuracy as the total success rate of a query for neighbors should be high, too. However, detection will sometimes report peers that are outside the AOI. This can be measured with the specifity which is high in case the retrieved results contain mostly correct peers.

5.2.2 Bandwidth

Another important metric is the bandwidth consumption on a per-peer basis and overall. It is important that the underlying networking system scales well. This means that bandwidth consumption rises virtually linearly with the number of peers. However, certain bandwidth peaks are expected in hot spot situations.

5.2.3 Spatial Position Difference

The spatial position difference is a metric that defines the deviation given the perceived position and the actual position. Mainly, the update frequency determines how often update messages are published and, thus, how big the deviation will be. Nevertheless, it is important for the NVE to have a preferably small deviation.

5.3 Experiment Setup

The simulation is performed in a specific environment which is described in the next sections. This includes the basic setup as well as different simulation modes.

5.3.1 Simulation Setup

The setup for the simulation is configured in the simulator's SQLite database. The basic setup for the simulation runs is as follows unless stated otherwise:

Bootstrap Node	100 MBit/s, always on, no churn, no delays
Normal Nodes	16/1 MBit/s ADSL2+, always on, no churn, 5 ms last hop delay, linear join
High-Update Interval	100 ms
Low-Update Interval	1000 ms
$Lambda$	3

Further settings are specified differently for each experiment and defined in the corresponding sections below. The simulations are run on a 64 Bit, 16 core multi processor server running on Debian 6.0 GNU/Linux. The server offers 64 GiB of main memory.

5.3.2 Simulation Modes

The simulation includes two modes. First, a *mobility model* exists to solely focus on the Pub/Sub overlay. Therefore, only simple movements in a limited two-dimensional plane are made. No other interaction between the peers is done. Hence, the quality of the overlay can be measured more precisely due to elimination of secondary influences. Second, a real simulation of a NVE is achieved by using the Planet π4 game with artificial bots. This bot mimics human behavior and follows the game rules. It is based on a state-machine and acts by analyzing the current environment and properties.

5.4 Simulation Results and Comparison

This section presents the obtained results from various simulation runs. Also, the developed approach is compared to other network layers which include pSense and traditional C/S. Note, that the pSense overlay was developed for use in a two-dimensional space. Obviously, adding another dimension increases the data sent through the network slightly. For that reason, a modified version of pSense, pSense3D, is used for the three-dimensional virtual world.

5.4.1 Mobility Model

This scenario uses the aforementioned mobility model where the focus is on evaluating the overlay. The simulation features 128 peers in a two-dimensional plane. The virtual world has a size of 1000 by 1000 units whereas the AOI of each peer is set to 100 units. The test is run for 15 minutes with the standard settings for peer behavior. BubbleStorm's λ is varied from one to three.

Figure 5.2 shows the active nodes for the simulation. Obviously, joining finishes shortly after 2 minutes of runtime. In this scenario no peer crashes are involved.

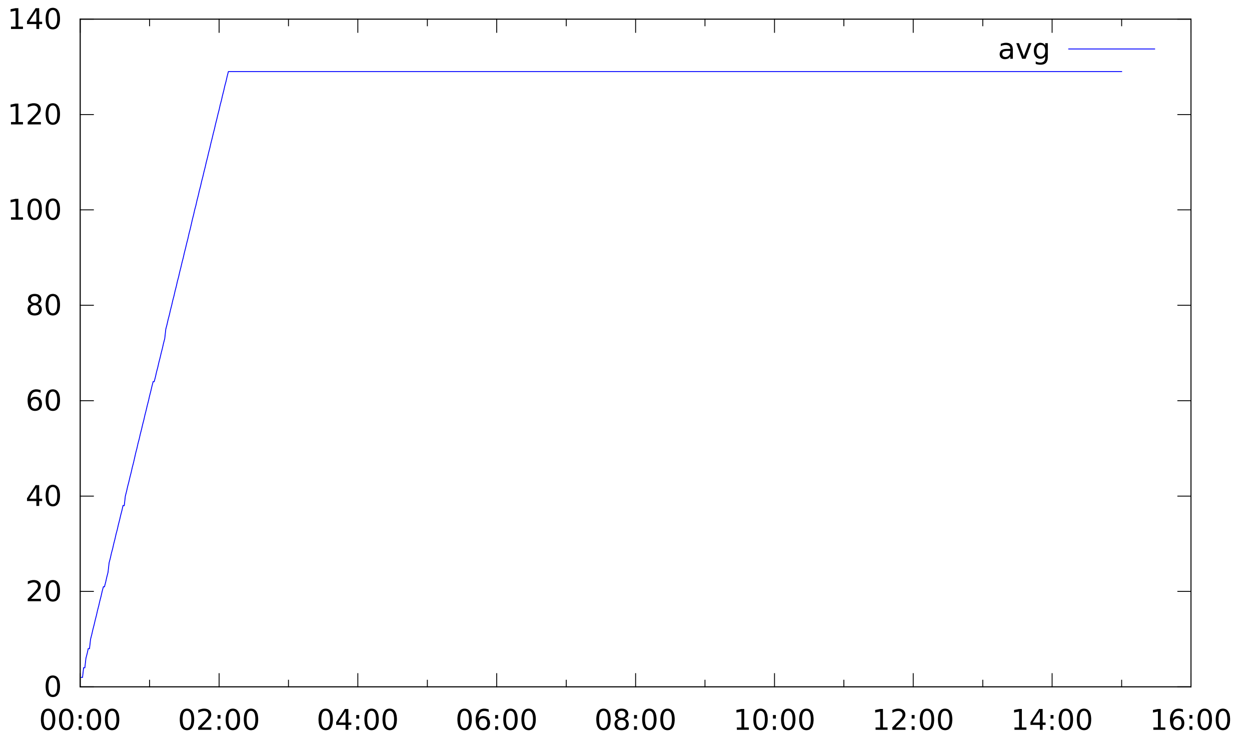


Figure 5.2: Running Peers for Mobility Model Simulations

Figure 5.3 and figure 5.4 show the total traffic consumption of all peers projected over the complete runtime. The first flat part of the graph is the joining phase which does not require high resources. After that, traffic increases linearly in both graphs. Finally, the difference in required total bandwidth between the λ values of one and three is around 15.7 MiB. This is due to reaching more nodes with one BubbleCast at a time. For comparison, a traditional C/S approach requires around 100 KiB whereas Planet π4 requires around 11 MiB. Note, that this scenario does not include churn and abrupt movements[1] and, thus, the maintenance overhead of pSense is low which results in a fairly lower traffic consumption.

Next, detection accuracy is evaluated in figure 5.5 and figure 5.6. Here, the developed Pub/Sub approach can outperform pSense with an average of 95.4 % compared to 86 %. With an accuracy of 88.2 % pSense3D performes somewhat better compared to its two-dimensional counterpart. Interestingly, changing λ to higher values only marginally changes the result with 95.5 % for λ of 3.0. This can be explained through the subscription cache which gets filled over time and delivers more results. Besides, the C/S approach has an accuracy of 97.7 %. Mainly, differences are influenced by the handling of peers that move outside of the AOI and trigger the timeout for soft and hard removal. These peers still receive updates in the current implementation but according to the metrics should not maintain connections anymore. Hence, they decrease accuracy. It should be noted, that this simulation was run with 5 seconds timeout

[1] Abrupt movements are present in Planet π4 after respawning at a randomly selected point.

values for the Pub/Sub overlay and 2 seconds for pSense which again proves that detection rates of this approach are even higher in comparison.

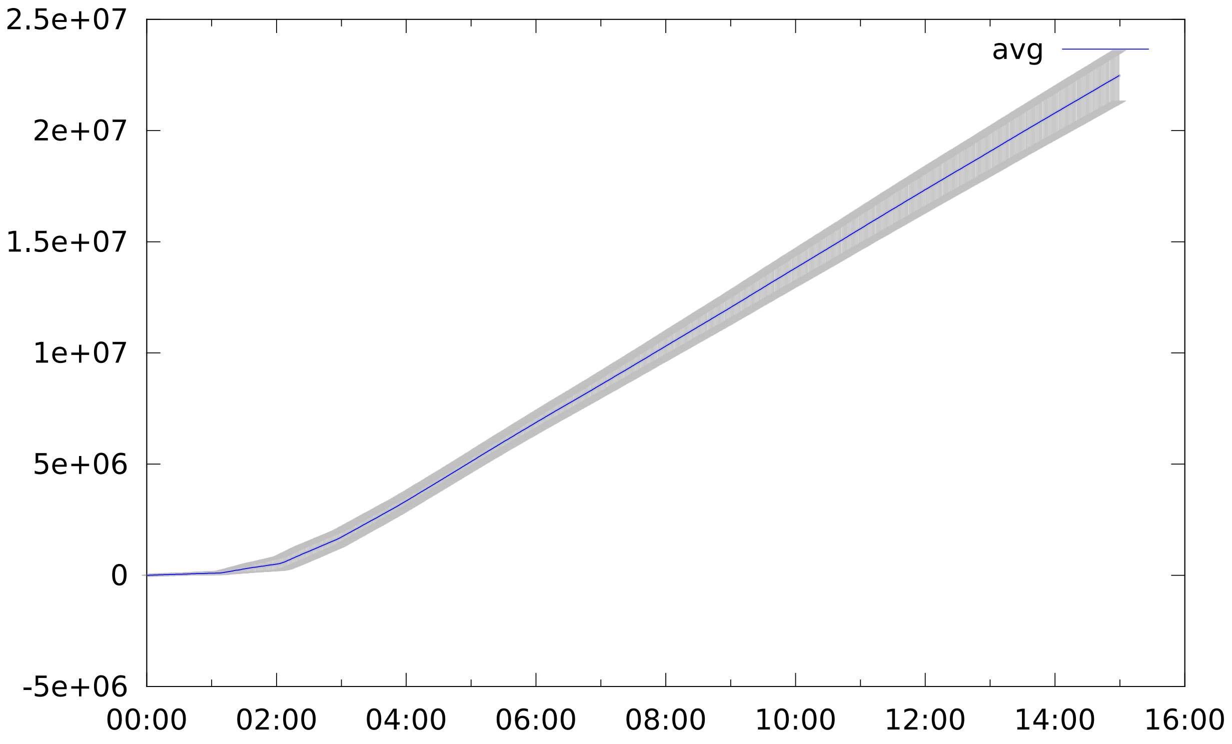


Figure 5.3: Total Traffic for Mobility Model; Pub/Sub with λ of 1.0

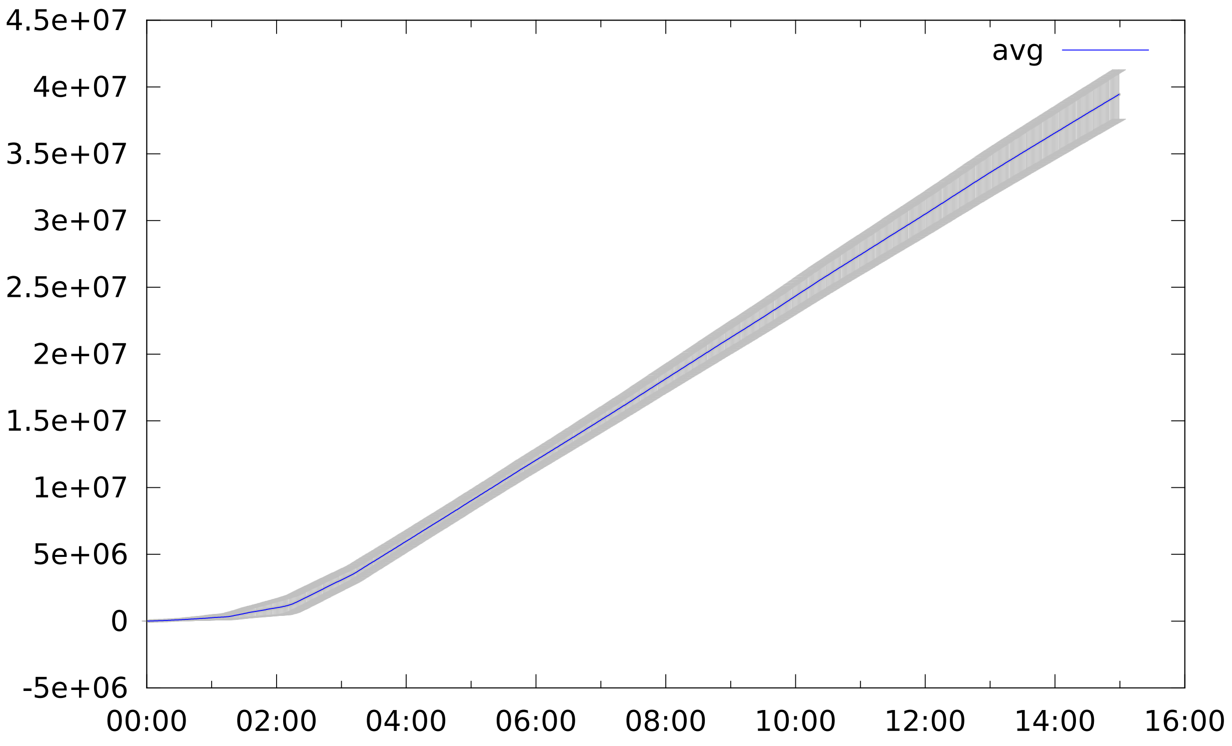


Figure 5.4: Total Traffic for Mobility Model; Pub/Sub with λ of 3.0

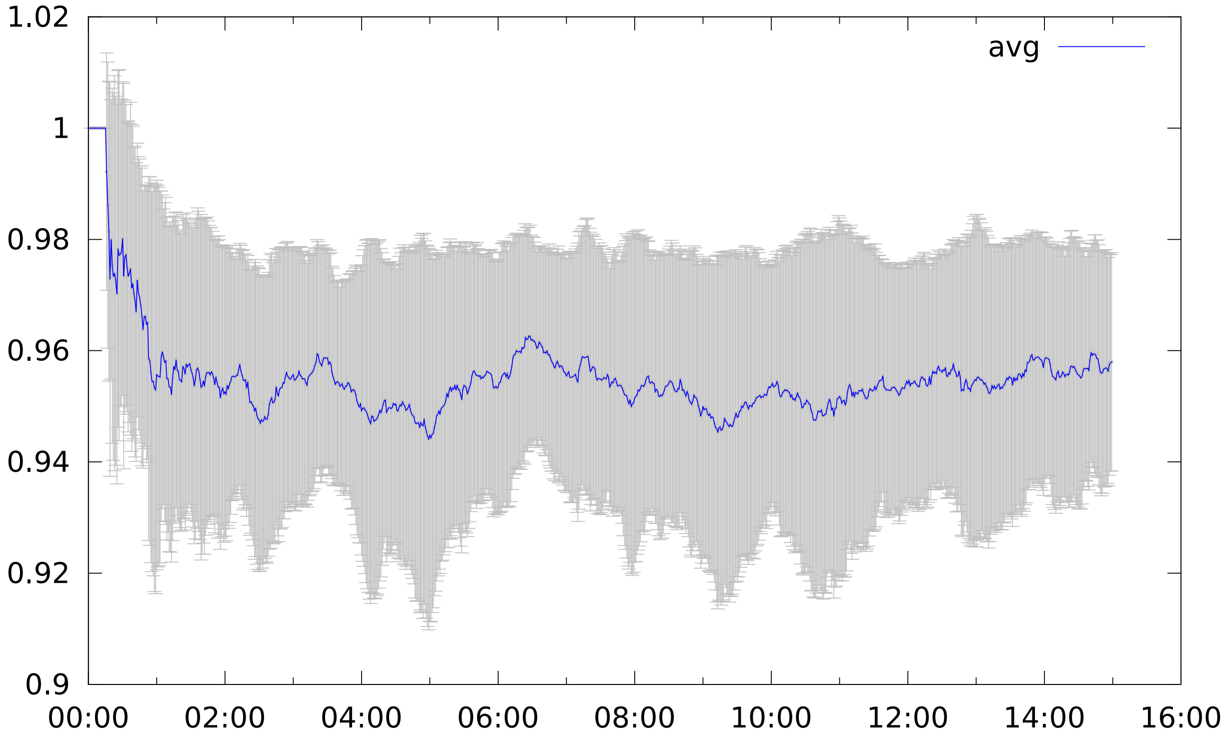


Figure 5.5: Detection Accuracy for Mobility Model; Pub/Sub with λ of 1.0

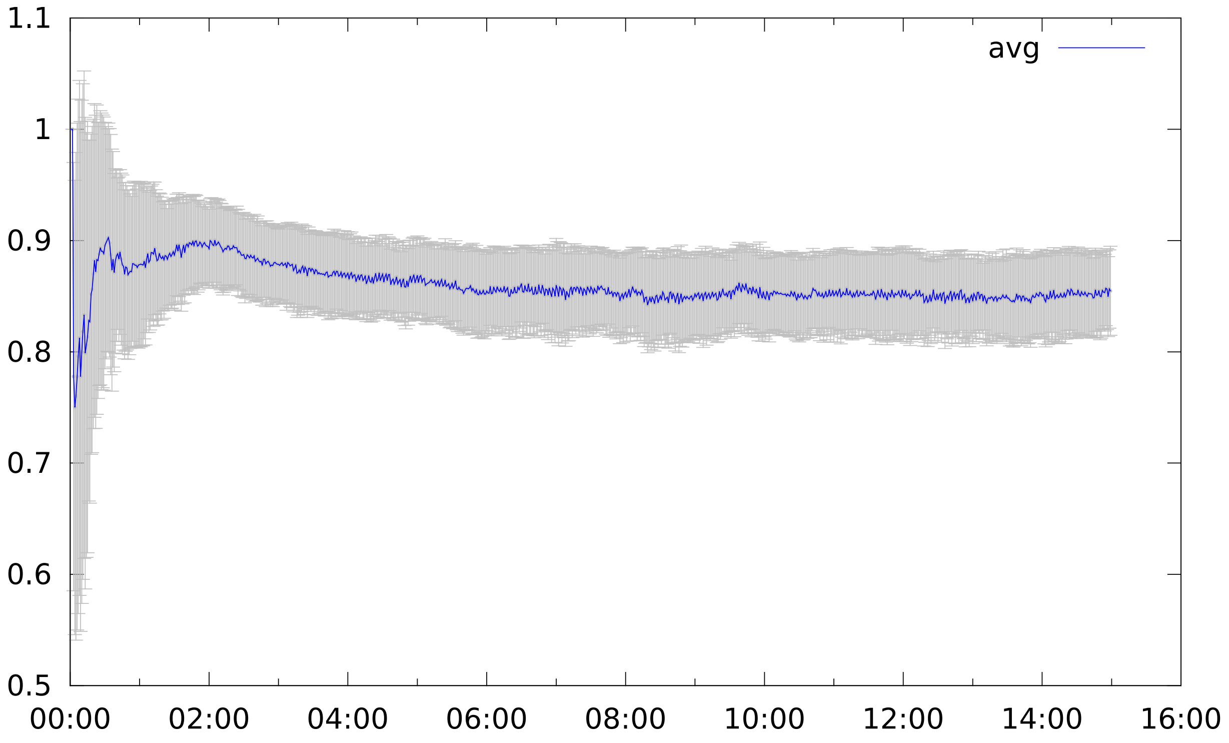


Figure 5.6: Detection Accuracy for Mobility Model; pSense

In combination with accuracy the specifity is evaluated. The graphs are shown in figures 5.7 and 5.8. Accordingly, the negative results detected by the Pub/Sub approach are more precise compared to pSense which means that not detected peers are most likely not of relevance.

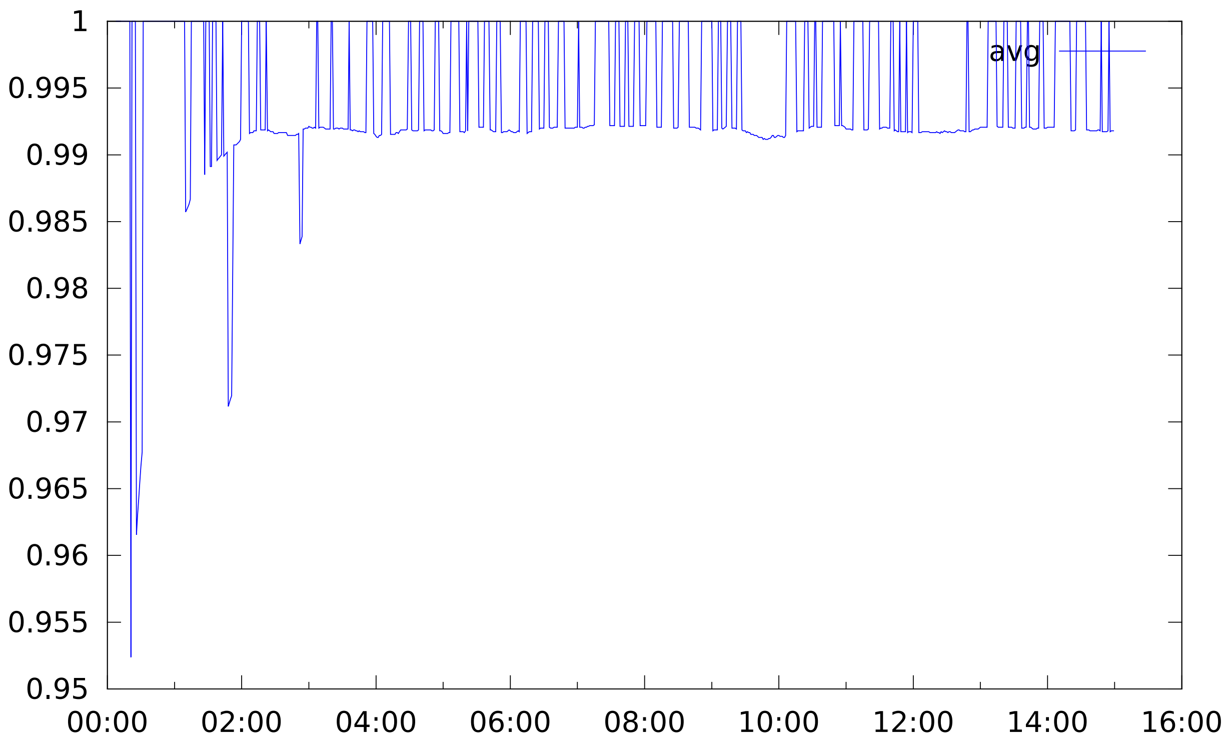


Figure 5.7: Detection Specifity for Mobility Model and Node 15; Pub/Sub with λ of 3.0

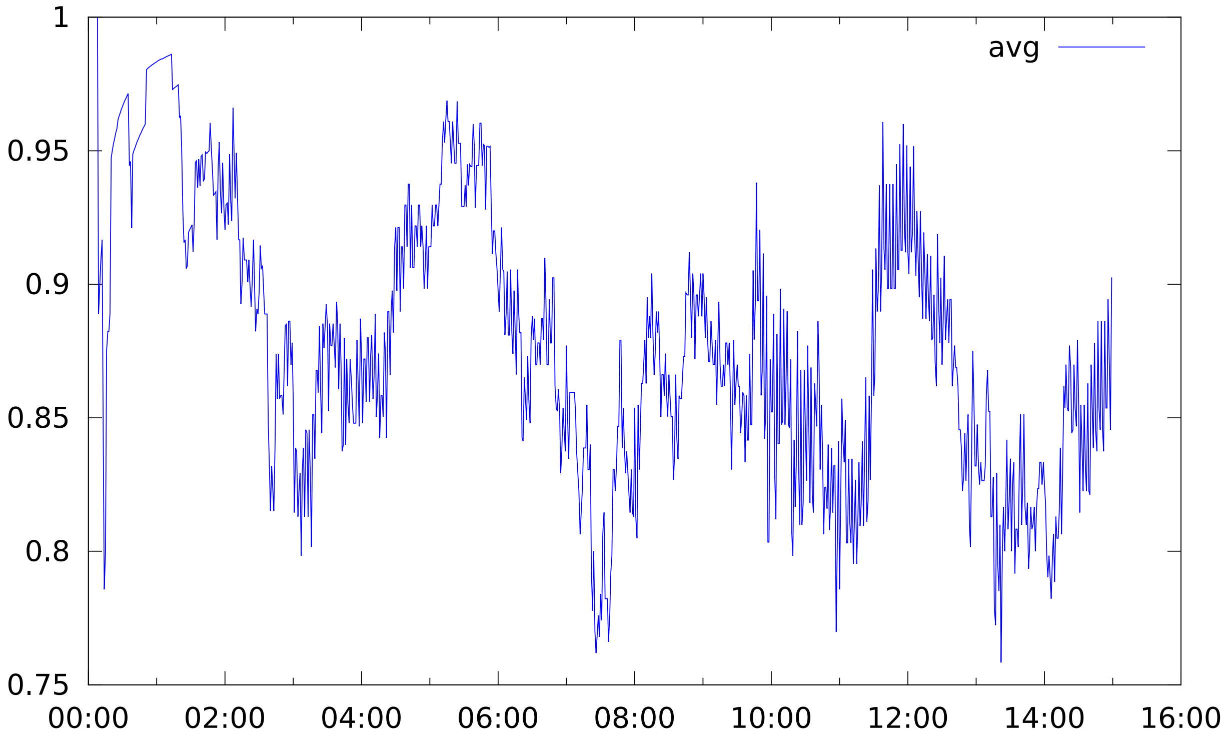


Figure 5.8: Detection Specifity for Mobility Model and Node 15; pSense

5.4.2 Planet $\pi4$

This scenario involves real game play with artificial bots. The simulation is executed with 32 peers for 10 minutes. Here, the joining phase takes about four minutes (see figure 5.9). Analogously, this scenario involves no peer crashes.

Most metrics behave similarly to the already introduced mobility model. However, some different observations can be made in the following metrics. Generally, detection accuracy drops in all P2P overlays to values between 76.1 % (Pub/Sub with λ of 1.0) and 77 % (pSense). This is due to the random respawning positions in case a player runs out of energy and, again, due to the soft timeouts. C/S achieves an expected higher accuracy of 90 %.

Next, totally used traffic now is lower using the Pub/Sub approach. Figures 5.10 and 5.11 show the course of the traffic curve. Here, pSense3D requires nearly 7 MiB more traffic not least because increased overlay maintenance. Again, traffic rises linearly in both overlays. In comparison, traditional C/S requires roughly 10 MiB of traffic.

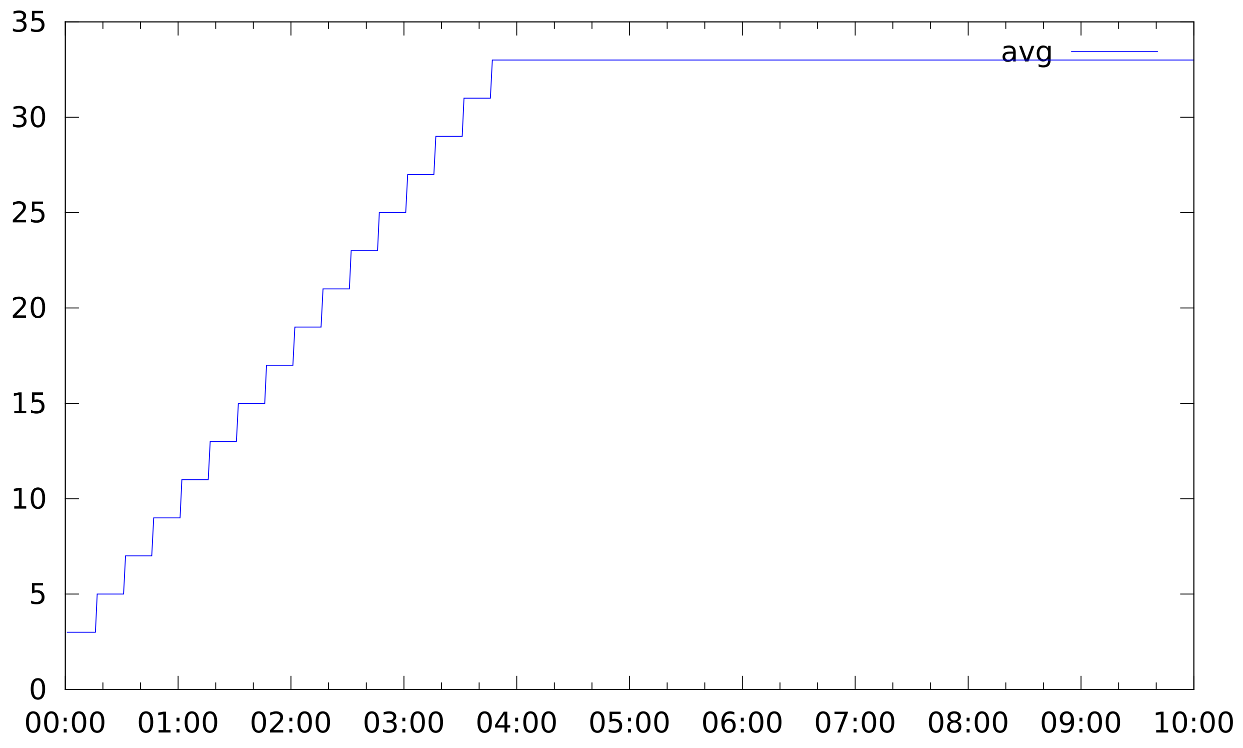


Figure 5.9: Running Peers for Planet $\pi 4$ Simulations

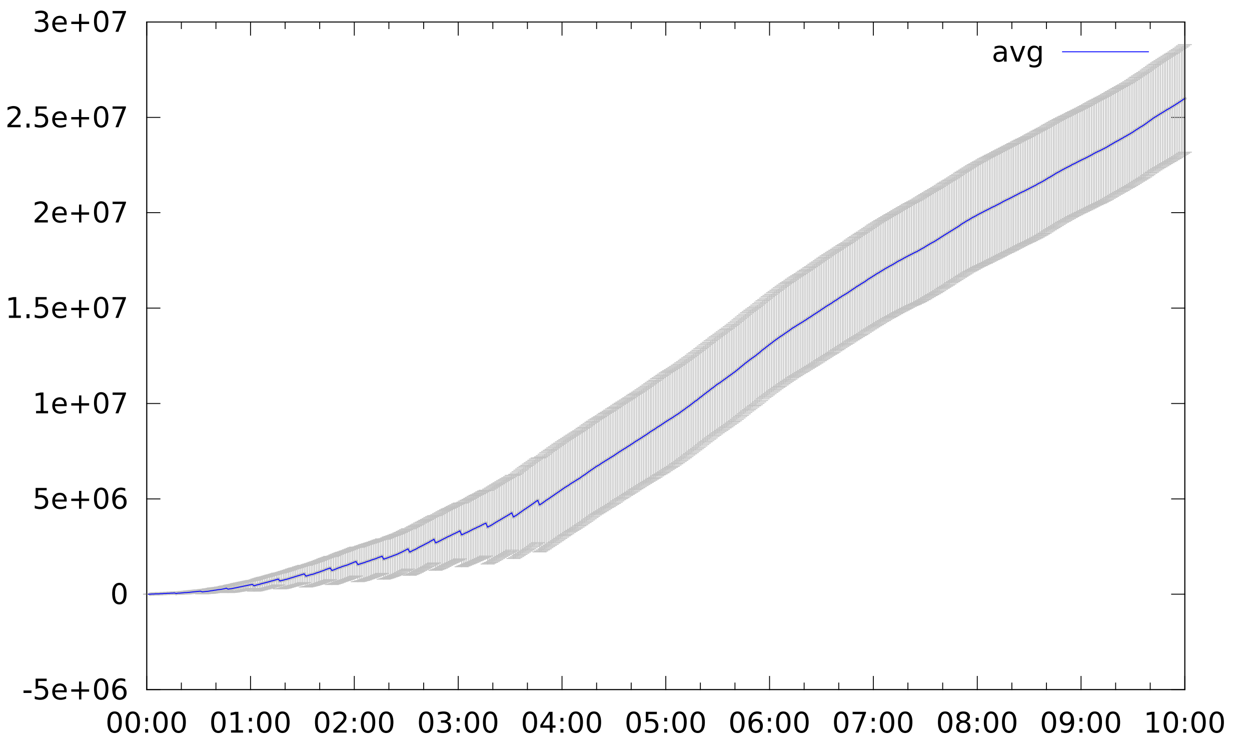


Figure 5.10: Total Traffic for Planet $\pi 4$; Pub/Sub with λ of 1.0

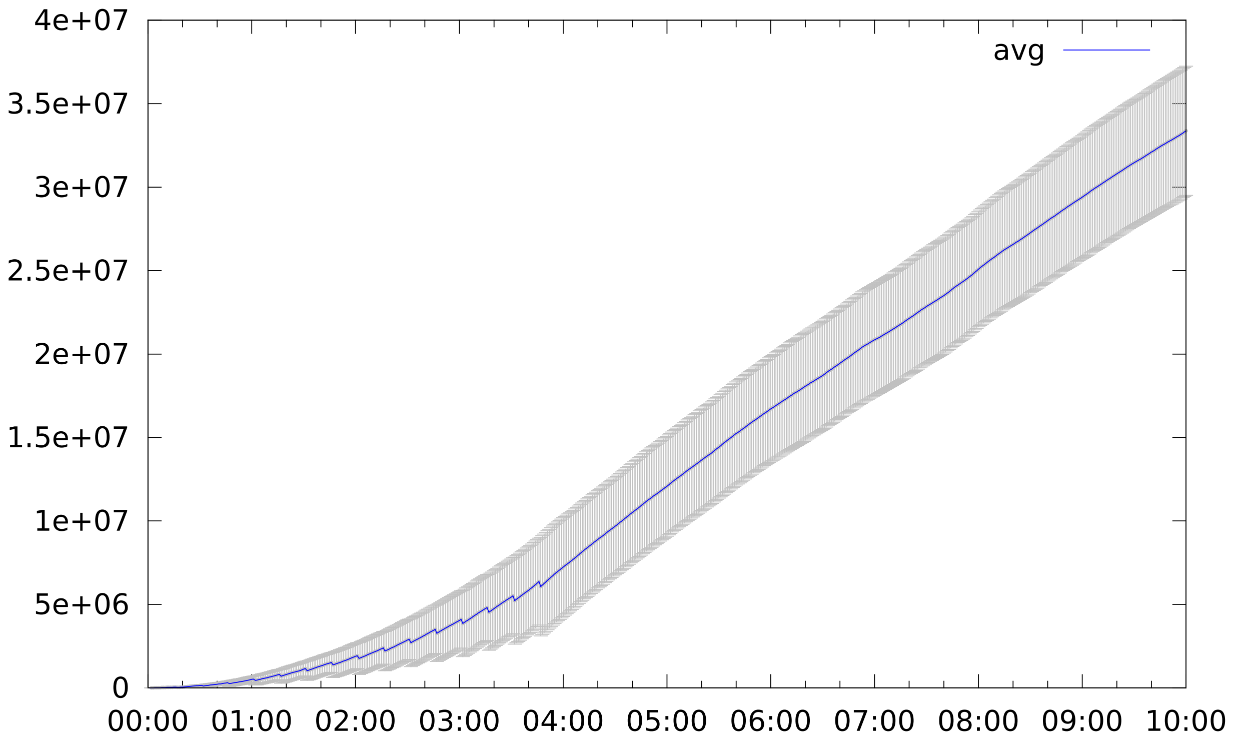


Figure 5.11: Total Traffic for Mobility Model; pSense3D

An interesting observation can be made in the Planet $\pi 4$ scenario for the number of known and "should be known" neighbors. Generally, variations are to be expected due to varying detection rates. However, the pSense/pSense3D implementations suffer from a too high connection degree even with lower timeout values. The Pub/Sub approach adapts better to changing neighbor sets.

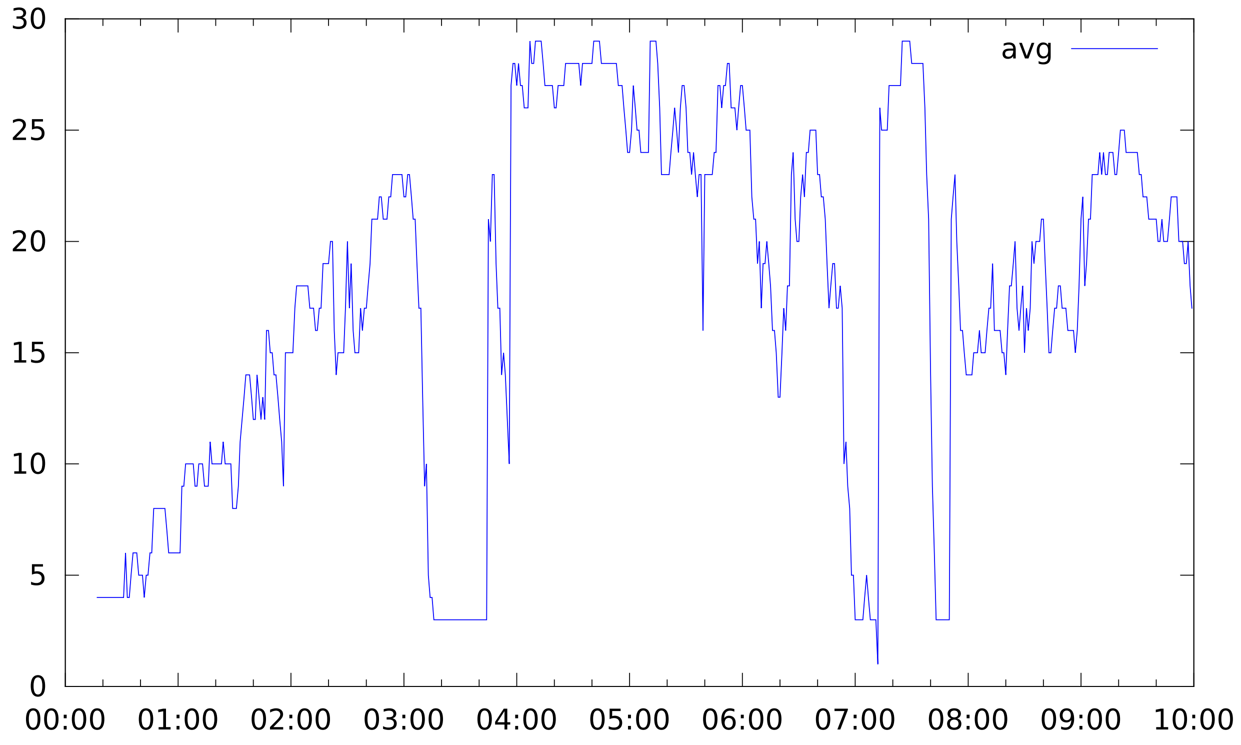


Figure 5.12: Number of “Should Be Known Neighbors” for Planet $\pi 4$ and Node 15; Pub/Sub with λ of 2.0

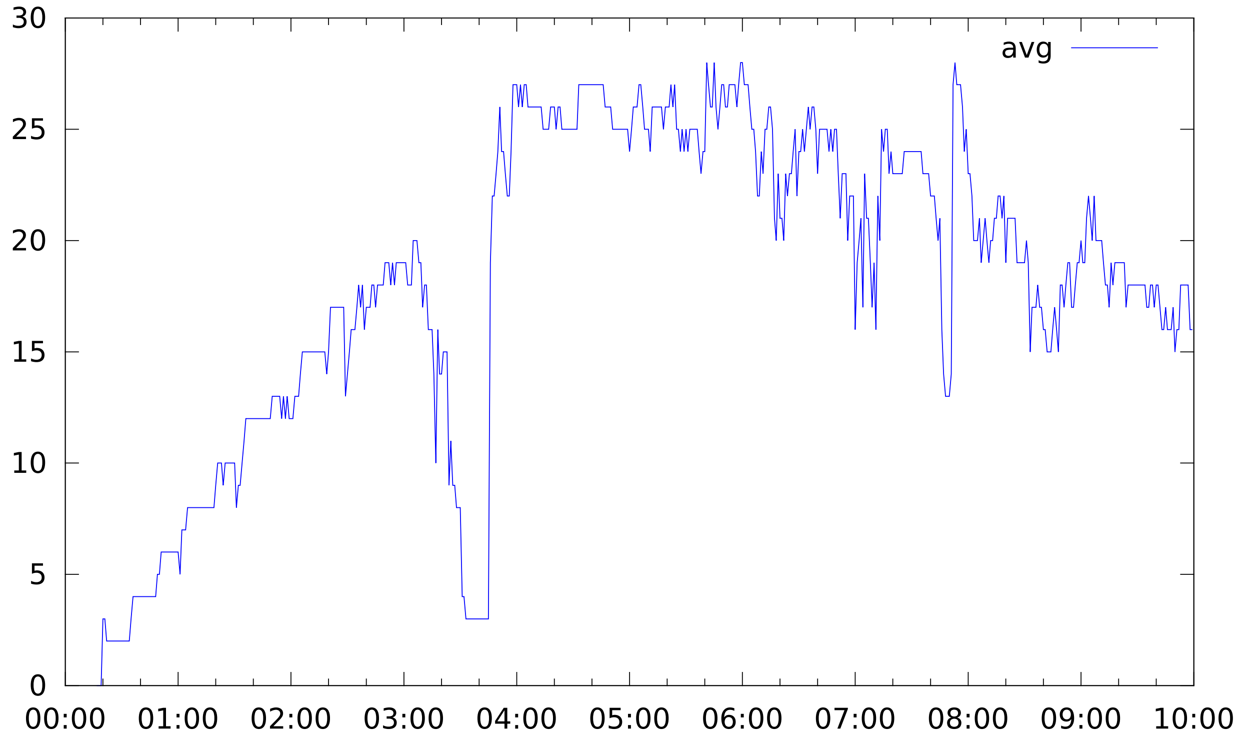


Figure 5.13: Number of “Known Neighbors” for Planet $\pi 4$ and Node 15; Pub/Sub with λ of 2.0

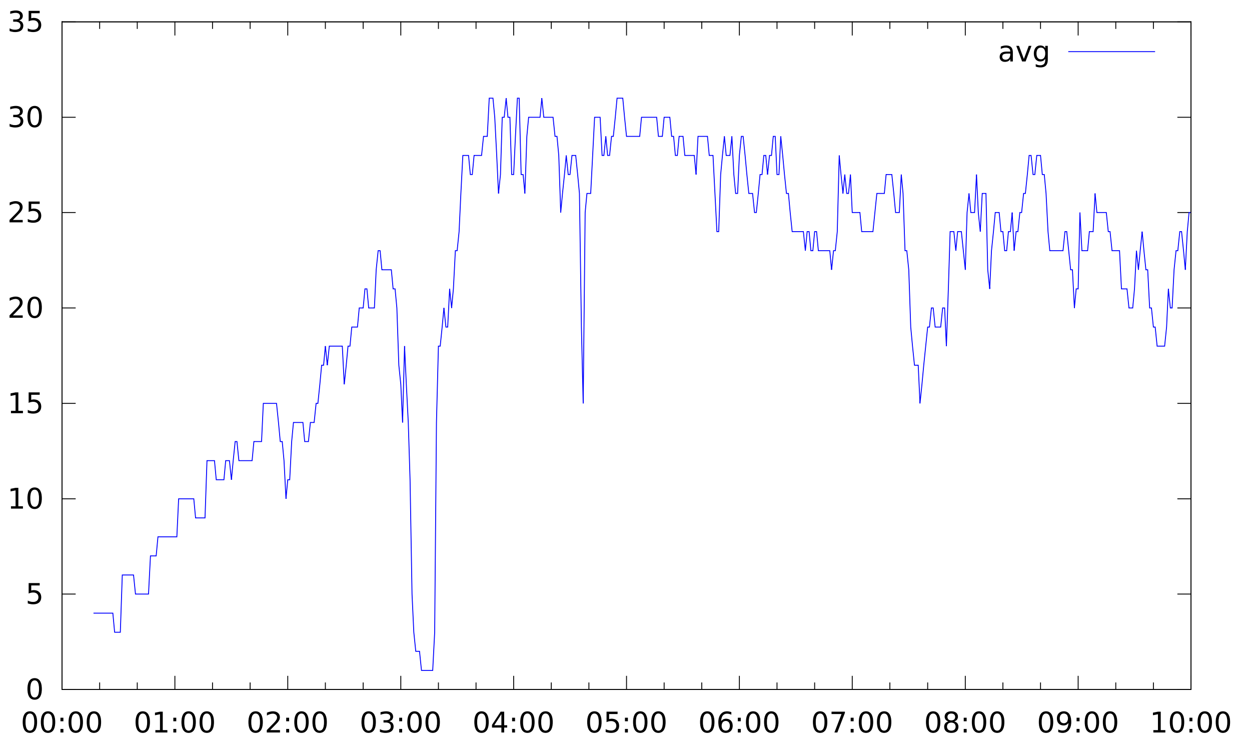


Figure 5.14: Number of “Should Be Known Neighbors” for Planet $\pi 4$ and Node 15; pSense

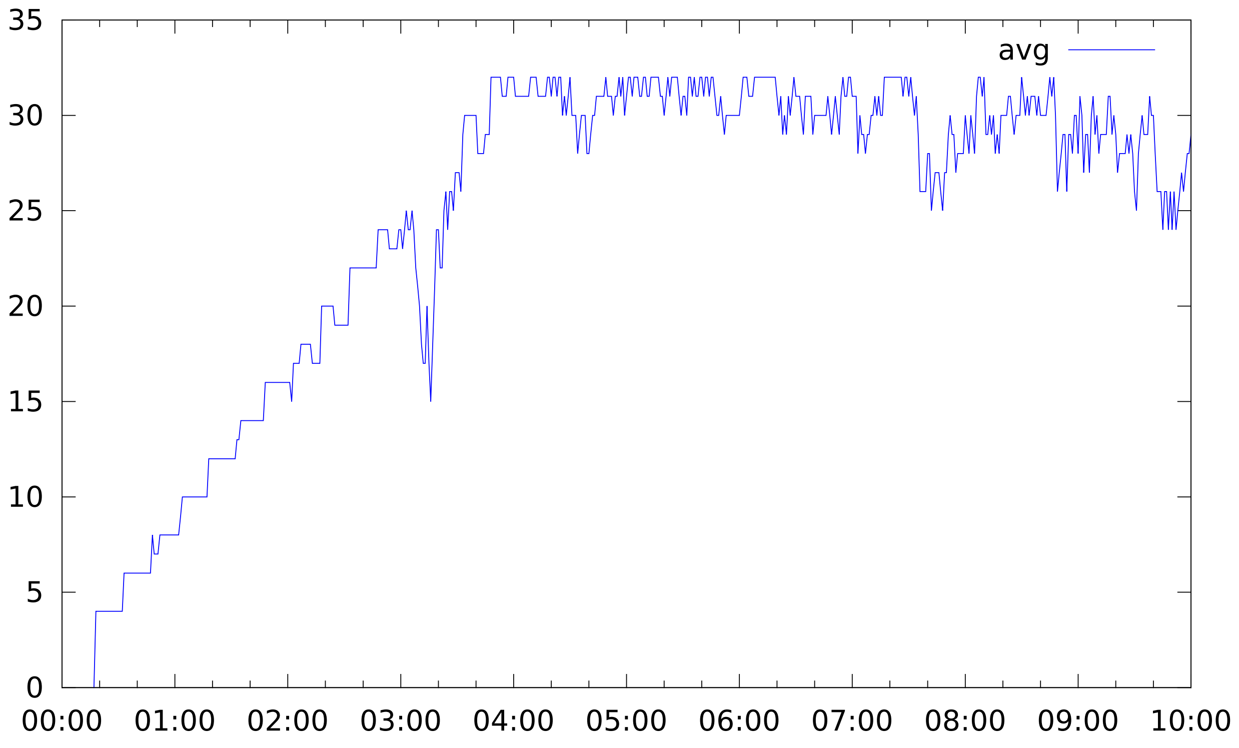


Figure 5.15: Number of “Known Neighbors” for Planet $\pi 4$ and Node 15; pSense

5.4.3 Summary

The evaluation proved that the developed Pub/Sub approach performs good in the area of NVE. Using movement data generated by artificial players demonstrated that under real world conditions Pub/Sub can outperform pSense in both traffic and handling of concurrent connections. The presented solution scales well and can generate precise enough results for unaffected game play experience. Interestingly, different λ values do not change the results greatly. As traffic however increases heavily between λ of one and three and detection results only varies slightly, the value of one is sufficient.

Due to not yet stable support for churn models in the simulator these simulations could not be run. However, the Pub/Sub overlay should perform very good as BubbleStorm handles high churn rates very well. Further, through the dissemination of subscriptions on nodes by BubbleCasts no maintenance is required in case of node crashes. The next interval will update the overlay with new subscriptions which can be used by others immediately. However, optimizations seem possible at the subscription cache mechanism to decrease traffic further.

Also, the time required for one simulation to finish can be optimized as the time is still high. The resources for one run with many peers are also in a magnitude that can be optimized to support a larger amount of simultaneous peers.

6 Conclusion

NVEs became a popular research topic in the last decade. This is due to famous MMOGs with thousands to millions of concurrent players. Managing such a vast amount of players goes beyond the practicality of pure C/S approaches in terms of server cost and maintenance. Hence, P2P based solutions were developed. The processing of interest management in NVEs is a fundamental part. As this includes subscribing to certain events in a player's AOI and publishing own positional updates, a Pub/Sub approach seems natural to apply.

The realization of a Pub/Sub mechanism in the area of NVEs has been the core task of this thesis. Contrary to many other P2P based Pub/Sub overlays, an unstructured P2P overlay was chosen for more flexibility. The features and advantages of BubbleStorm were leveraged to create a scalable and robust Pub/Sub overlay. This enabled a more flexible way of subscription handling as all queryable languages are supported. The developed Pub/Sub system features area subscriptions and point publications. Subscriptions are distributed in the network based on the BubbleCast mechanism that achieves good network reach, fast execution, and scalability. The developed system further minimizes network transport costs by introducing various caching mechanisms. Publications, i. e., positional updates are disseminated using directly connected end points. Because of the dynamic nature of NVEs with steadily changing neighbors, elaborate message dissemination techniques would suffer from high latency and maintenance overhead and were not applied. Further, C++ language bindings for BubbleStorm were implemented to be able to use the existing API written in SML. This results in high dependencies during the build process. Decreased latency in connection setup times could be achieved by applying the CUSP protocol. The channel/stream concept fits well for NVE applications that require many simultaneous connections. However, CUSP programming in C++ introduces some challenges to already non-trivial network programming.

Planet π4 was used to measure the feasibility and quality of the developed Pub/Sub overlay. The open world scenario in combination with the ability to support plenty of concurrent players makes it a good choice for working with the Pub/Sub system. Adaptions were made to use this new overlay as the network engine. Also, a mechanism to collect statistics on a per-peer as well as global base were implemented as part of this thesis. Metrics include precision and recall as well as bandwidth consumption, detection rates, and position accuracy.

The evaluation of the developed Pub/Sub overlay was done programmatically by discrete event simulation. The overlay was simulated with a simplified mobility model and the more complex Planet π4 application. Results showed a good and stable accuracy and linear bandwidth consumption. In comparison to pSense, the results are slightly better in Planet π4 traffic behavior and detection accuracy in the mobility model. However, the approaches differ fundamentally and yet can show good applicability for NVEs. Moreover, the Pub/Sub approach handles churn better as it does not need to maintain a complex overlay structure as pSense which is one of the reasons for a lower total traffic in the Planet π4 simulation. In addition, further optimizations are possible and can increase overlay quality.

Concluding, the developed approach proved to work well for MMOGs and most likely other scenarios too. Scalability, reliability, and, more importantly, latency of the overlay and thus the interest management are satisfying and do not affect the positive user experience. Moreover, the approach is not limited to the area of virtual environments. Subscribing to complex events is easily possible and supported by the overlay as evaluating a query is handled on a peer. Admittedly, lowest-latency delivery of publications is not necessary in every application. Yet, this is part of the application and can be changed according to the requirements. Undoubtedly, many applications can benefit from this Pub/Sub overlay where scalability and reliability are the main demands.

7 Outlook

This thesis mainly focused on scalability and successful interest management in the area of MMOGs. Typically, multiplayer games are prone to cheating and require appropriate measures to prevent tampering. Such security features can be integrated into the Pub/Sub overlay to prevent wittingly or unwittingly alteration of messages be it malicious behavior or message corruption. As a beginning, CUSP's encryption functionality could be applied to ensure confidentiality of exchanged messages. Further, authentication measures could improve overall security greatly. The impact of such measures in terms of overlay scalability and reliability would be an interesting research topic.

Further, the detection of far-away neighbors could be improved. Thus, the AOI could be separated into near, medium, and far regions. Near peers would receive high-frequency updates whereas medium and far peers would have considerably lower update intervals. This would improve the detection speed of fast incoming peers from a distant position as well as traffic consumption.

A way to decrease peer traffic would be the integration of a multicast approach for message dissemination instead of directly connected end points. Many ALM (Application Layer Multicast) systems already exist in various forms. In general, these systems can be categorized into structured and probabilistic approaches, that is, systems that try to propagate messages by routing them through the created overlay or systems that disseminate messages using forwarding to randomly selected neighbors and stochastic models. The latter is commonly referred to as gossiping (Allani et al., 2009). Usually, structured ALM require overlay maintenance which is particularly bad in high churn scenarios like MMOGs. Also, applying ALM techniques comes at the high risk of severely increased latency which is particularly unfavorable in MMOGs.

Of course, NVEs are not the only possible applications that involve plenty of participants. For example, wireless sensor networks with plenty of nodes could leverage the potential of Pub/Sub and a scalable infrastructure. Also, scenarios in the area of transportation are imaginable where cars can broadcast information concerning the current flow of traffic and, for example, warn about traffic jams. A fast and reliable Pub/Sub system would prove advantageous.

Concluding, the area of P2P-based Pub/Sub systems is valuable and surely will gain more importance in upcoming distributed worlds.

Bibliography

[Aberer et al. 2003] Aberer, Karl ; Cudré-Mauroux, Philippe ; Datta, Anwitaman ; Despotovic, Zoran ; Hauswirth, Manfred ; Punceva, Magdalena ; Schmidt, Roman: P-Grid: a self-organizing structured P2P system. In: *ACM SIGMOD Record* 32 (2003), Nr. 3, p. 29–33. – URL http://portal.acm.org/citation.cfm?id=945729. – ISSN 0163-5808

[Allani et al. 2009] Allani, Mouna ; Garbinato, Benoît ; Pedone, Fernando: *Application Layer Multicast*. Chap. 9, p. 191–218. In: Garbinato, Benoît (Editor) ; Miranda, Hugo (Editor) ; Rodrigues, Luís (Editor): *Middleware for Network Eccentric and Mobile Applications*. Heidelberg : Springer Berlin Heidelberg, 2009. – ISBN 978-3-540-89706-4

[Bharambe et al. 2008] Bharambe, Ashwin R. ; Douceur, John R. ; Lorch, Jacob R. ; Moscibroda, Thomas ; Pang, Jeffrey ; Seshan, Srinivasan ; Zhuang, Xinyu: Donnybrook: Enabling Large-Scale, High-Speed, Peer-to-Peer Games. In: *ACM SIGCOMM Computer Communication Review* 38 (2008), october, Nr. 4, p. 389—-400. – URL http://portal.acm.org/citation.cfm?doid=1402946.1403002. – ISSN 01464833

[Bharambe et al. 2006] Bharambe, Ashwin R. ; Pang, Jeffrey ; Seshan, Srinivasan: Colyseus : A Distributed Architecture for Online Multiplayer Games. In: *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*. San Jose, CA : USENIX Association, 2006

[Bharambe et al. 2002] Bharambe, Ashwin R. ; Rao, Sanjay ; Seshan, Srinivasan: Mercury: a scalable publish-subscribe system for internet games. In: *Network and System Support for Games* (2002). – URL http://portal.acm.org/citation.cfm?id=566500.566501

[Botev et al. 2008] Botev, Jean ; Hohfeld, Alexander ; Schloss, Hermann ; Scholtes, Ingo ; Sturm, Peter ; Esch, Markus: The HyperVerse: concepts for a federated and Torrent-based '3D Web'. In: *International Journal of Advanced Media and Communication* 2 (2008), Nr. 4, p. 331–350. – URL http://inderscience.metapress.com/index/J840K59785686367.pdf. – ISSN 1462-4613

[Carzaniga et al. 2000] Carzaniga, Antonio ; Rosenblum, David S. ; Wolf, Alexander L.: Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In: *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. Portland, Oregon, United States : ACM New York, NY, USA, 2000, p. 219–227. – URL http://portal.acm.org/citation.cfm?id=343477.343622. – ISBN 1-58113-183-6

[Carzaniga et al. 2003] Carzaniga, Antonio ; Rutherford, Matthew J. ; Wolf, Alexander L.: A routing scheme for content-based networking. In: *IEEE Infocom 2004* (2003), p. 918–928. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1356979. ISBN 0-7803-8355-9

[Douglas et al. 2005] Douglas, Scott ; Tanin, Egemen ; Harwood, Aaron ; Karunasekera, Shanika: Enabling massively multi-player online gaming applications on a P2P architecture. In: *Proceedings of the IEEE international conference on information and automation*, Citeseer, 2005, p. 7–12. – URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.94.2225

[Esch et al. 2008] Esch, Markus ; Botev, Jean ; Schloss, Hermann ; Scholtes, Ingo: GP3 - A Distributed Grid-based Spatial Index Infrastructure for Massive Multiuser Virtual Environments. In: *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, IEEE Computer Society, 2008, p. 811–816

[Eugster et al. 2003] Eugster, Patrick T. ; Felber, Pascal A. ; Guerraoui, Rachid ; Kermarrec, Anne-Marie: The Many Faces of Publish/Subscribe. In: *ACM Computing Surveys (CSUR)* 35 (2003), Nr. 2, p. 114–131. – URL http://portal.acm.org/citation.cfm?id=857076.857078. – ISSN 0360-0300

[Hu 2009] Hu, Shun-Yun: Spatial Publish Subscribe. In: *IEEE Virtual Reality 2009*. Lafayette, Louisiana, USA, 2009. – URL http://pap.vs.uni-due.de/MMVE09/papers/p8.pdf

[Hu et al. 2008] Hu, Shun-Yun ; Chang, Shao-Chen ; Jiang, Jehn-Ruey: Voronoi State Management for Peer-to-Peer Massively Multiplayer Online Games. In: *2008 5th IEEE Consumer Communications and Networking Conference* (2008), p. 1134–1138. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4446553. ISBN 1-4244-1457-1

[Hu et al. 2006] Hu, Shun-Yun ; Chen, Jui-Fa ; Chen, Tsu-Han: VON: a scalable peer-to-peer network for virtual environments. In: *IEEE Network* 20 (2006), july, Nr. 4, p. 22–31. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1668400. – ISSN 0890-8044

[Hu and Liao 2004] Hu, Shun-Yun ; Liao, Guan-Ming: Scalable peer-to-peer networked virtual environment. In: *Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04 Network and system support for games - SIGCOMM 2004 Workshops*. New York, New York, USA : ACM Press, 2004, p. 129. – URL http://portal.acm.org/citation.cfm?doid=1016540.1016552. – ISBN 158113942X

[Hu et al. 2010] Hu, Shun-Yun ; Wu, Chuan ; Buyukkaya, Eliya ; Chien, Chien-hao ; Lin, Tzu-Hao ; Abdallah, Maha ; Jiang, Jehn-Ruey ; Chen, Kuan-Ta: A Spatial Publish Subscribe Overlay for Massively Multiuser Virtual Environments. In: *2010 International Conference on Electronics and Information Engineering* Volume 2. Kyoto, Japan : IEEE, august 2010, p. V2–314–V2–318. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5559789. – ISBN 978-1-4244-7679-4

[Jiang et al. 2008] Jiang, Jehn-Ruey ; Huang, Yu-Li ; Hu, Shun-Yun: Scalable AOI-Cast for Peer-to-Peer Networked Virtual Environments. In: *The 28th International Conference on Distributed Computing Systems Workshops*, IEEE Computer Society, june 2008, p. 447–452. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4577825. – ISBN 978-0-7695-3173-1

[Kang et al. 2010] Kang, InSung ; Choi, SungJin ; Jung, SoonYoung ; Lee, SangKeun: Tree-Based Index Overlay in Hybrid Peer-to-Peer Systems. In: *Journal of Computer Science and Technology* 25 (2010), march, Nr. 2, p. 313–329. – URL http://www.springerlink.com/index/10.1007/s11390-010-9326-0. – ISSN 1000-9000

[Knutsson et al. 2004] Knutsson, Björn ; Lu, Honghui ; Xu, Wei ; Hopkins, Bryan: Peer-to-peer support for massively multiplayer games. In: *IEEE INFOCOM* (2004)

[Manning et al. 2009] Manning, Christopher D. ; Raghavan, Prabhakar ; Schütze, Hinrich: *An introduction to information retrieval*. Cambridge, England : Cambridge University Press, 2009 (c). – 581 p. – ISBN 0521865719

[Maymounkov and Mazieres 2002] Maymounkov, Petar ; Mazieres, David: *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*. Volume 2429. p. 53–65. In: Druschel, Peter (Editor) ; Kaashoek, Frans (Editor) ; Rowstron, Antony (Editor): *Peer-to-Peer Systems* Volume 2429, Springer, 2002. – URL http://www.springerlink.com/index/2EKX2A76PTWD24QT.pdf

[Michael et al. 2007] Michael, Maged ; Moreira, Jose E. ; Shiloach, Doron ; Wisniewski, Robert W.: Scale-up x Scale-out: A Case Study using Nutch/Lucene. In: *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), p. 1—-8. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4228359. ISBN 1-4244-0909-8

[Miller and Crowcroft 2010] Miller, J.L. ; Crowcroft, Jon: The Near-Term Feasibility of P2P MMOGs. In: *Proc. of International Workshop on Network and Systems Support for Games (NetGames)*, URL `http://research.microsoft.com/pubs/140999/NetGames-P2PMMOG.pdf`, 2010

[Miller 2009] Miller, Richard: *WoW's Back End: 10 Data Centers, 75,000 Cores*. November 2009. – URL `http://www.datacenterknowledge.com/archives/2009/11/25/wows-back-end-10-data-centers-75000-cores/`. – Accessed on June 23, 2010

[Milner et al. 1997] Milner, Robin ; Tofte, Mads ; Harper, Robert ; MacQueen, David: *The Definition of Standard ML*. Revised Ed. Cambridge, MA, USA : The MIT Press, 1997. – 128 p. – ISBN 978-0-262-63181-5

[Neumann et al. 2007] Neumann, Christoph ; Prigent, Nicolas ; Varvello, Matteo ; Suh, Kyoungwon: Challenges in Peer-to-Peer Gaming. In: *ACM SIGCOMM Computer Communication Review* 37 (2007), january, Nr. 1, p. 79–82. – URL `http://portal.acm.org/citation.cfm?doid=1198255.1198269`. – ISSN 01464833

[Pang et al. 2007] Pang, Jeffrey ; Uyeda, Frank ; Lorch, Jacob R.: Scaling peer-to-peer games in low-bandwidth environments. In: *Proc. 6th Intl. Workshop on Peer-to-Peer* (2007). – URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.6210&rep=rep1&type=pdf`

[Pantel and Wolf 2002] Pantel, Lothar ; Wolf, Lars C.: *On the suitability of dead reckoning schemes for games*. p. 79–84. In: *Proceedings of the 1st workshop on Network and system support for games - NETGAMES '02*. New York, New York, USA : ACM Press, 2002. – URL `http://portal.acm.org/citation.cfm?doid=566500.566512`. – ISBN 1581134932

[Perng et al. 2004] Perng, Ginger ; Wang, Chenxi ; Reiter, Michael K.: Providing content-based services in a peer-to-peer environment. In: *International Workshop on Distributed Event-based Systems - 26th International Conference on Software Engineering* 2004 (2004), Nr. 918, p. 74–79. – URL `http://link.aip.org/link/IEESEM/v2004/i918/p74/s1&Agg=doi`. ISBN 0 86341 433 8

[Rahimian et al. 2011] Rahimian, Fatemeh ; Girdzijauskas, Sarunas ; Payberah, A.H. ; Haridi, Seif: Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe. In: *International Parallel & Distributed Processing Symposium IPDPS 2011*. Anchorage, Alaska, USA : IEEE Computer Society, 2011. – URL `http://soda.swedish-ict.se/4087/`

[Ratnasamy et al. 2001] Ratnasamy, Sylvia ; Francis, Paul ; Handley, Mark ; Karp, Richard ; Shenker, Scott: A scalable content-addressable network. In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. San Diego, USA : ACM, 2001, p. 161–172. – URL `http://portal.acm.org/citation.cfm?id=383072`. – ISBN 1581134118

[Rowstron and Druschel 2001] Rowstron, Antony ; Druschel, Peter: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. Heidelberg, Germany, 2001, p. 329—-350. – URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.5987`

[Rowstron et al. 2001] Rowstron, Antony ; Kermarrec, Anne-Marie ; Castro, Miguel ; Druschel, Peter: *Scribe: The Design of a Large-Scale Event Notification Infrastructure*. p. 30–43. In: Crowcroft, Jon (Editor) ; Hofmann, Markus (Editor): *Networked Group Communication* Volume 2233. Berlin, Heidelberg : Springer Berlin / Heidelberg, october 2001. – URL `http://www.springerlink.com/index/10.1007/3-540-45546-9`. – ISBN 978-3-540-42824-4

[Schmieg et al. 2008] Schmieg, Arne ; Stieler, Michael ; Jeckel, Sebastian ; Kabus, Patric ; Kemme, Bettina ; Buchmann, Alejandro P.: 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*. Los Alamitos, CA, USA : IEEE, september 2008, p. 247–256. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4627287. – ISBN 978-0-7695-3318-6

[Shen et al. 2010] Shen, Xuemin ; Yu, Heather ; Buford, John ; Akon, Mursalin ; Shen, Xuemin (Editor) ; Yu, Heather (Editor) ; Buford, John (Editor) ; Akon, Mursalin (Editor): *Handbook of Peer-to-Peer Networking*. Springer-Verlag New York, 2010. – ISBN 9780387097503

[Stoica et al. 2001] Stoica, Ion ; Morris, Robert ; Karger, David ; Kaashoek, M. F. ; Balakrishnan, Hari: Chord: A scalable peer-to-peer lookup service for internet applications. In: *SIGCOMM Computer Communication Review* 31 (2001), Nr. 4, p. 149–160. – URL http://portal.acm.org/citation.cfm?id=964723.383071. ISBN 1581134118

[Tanin et al. 2006] Tanin, Egemen ; Harwood, Aaron ; Samet, Hanan: Using a Distributed Quadtree Index in Peer-to-Peer Networks. In: *The VLDB Journal* 16 (2006), april, Nr. 2, p. 165–178. – URL http://www.springerlink.com/index/10.1007/s00778-005-0001-y. – ISSN 1066-8888

[Terpstra et al. 2007a] Terpstra, Wesley W. ; Kangasharju, Jussi ; Leng, Christof ; Buchmann, Alejandro P.: Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In: *ACM SIGCOMM Computer Communication Review* 37 (2007), Nr. 4. – URL http://portal.acm.org/citation.cfm?id=1282387. – ISSN 0146-4833

[Terpstra et al. 2007b] Terpstra, Wesley W. ; Leng, Christof ; Buchmann, Alejandro P.: BubbleStorm: Analysis of Probabilistic Exhaustive Search in a Heterogeneous Peer-to-Peer System / Technische Universität Darmstadt. october 2007 (4). – Research Report. – ISSN 01464833

[Terpstra et al. 2010] Terpstra, Wesley W. ; Leng, Christof ; Lehn, Max ; Buchmann, Alejandro: Channel-based Unidirectional Stream Protocol (CUSP). In: *2010 Proceedings IEEE INFOCOM* (2010), march, p. 1–5. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5462238. ISBN 978-1-4244-5836-3

[Triebel et al. 2008] Triebel, Tonio ; Guthier, Benjamin ; Süselbeck, Richard ; Schiele, Gregor ; Effelsberg, Wolfgang: Peer-to-peer infrastructures for games. In: *NOSSDAV '08: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. New York, NY, USA : ACM, 2008, p. 123–124. – ISBN 978-1-60558-157-6

[Yu and Vuong 2005] Yu, Anthony P. ; Vuong, Son T.: 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, p. 104. – URL http://portal.acm.org/citation.cfm?id=1066007