INTEREST-BASED FILTERING OF SOCIAL DATA IN DECENTRALIZED ONLINE SOCIAL NETWORKS

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Udeep Tandukar

ABSTRACT

In Online Social Networks (OSNs) users are overwhelmed with huge amount of social data, most of which are irrelevant to their interest. Due to the fact that most current OSNs are centralized, people are forced to share their data with the site, in order to be able to share it with their friends, and thus they lose control over it. Decentralized Online Social Networks have been proposed as an alternative to traditional centralized ones (such as Facebook, Twitter, Google+, etc.) to deal with privacy problems and to allow users to maintain control over their data.

This thesis presents a novel peer-to-peer architecture for decentralized OSN and a mechanism that allows each node to filter out irrelevant social data, while ensuring a level of serendipity (serendipitous are social data which are unexpected since they do not belong in the areas of interest of the user but are desirable since they are important or popular). The approach uses feedback from recipient users to construct a model of different areas of interest along the relationships between sender and receiver, which acts as a filter while propagating social data in this area of interest. The evaluation of the approach, using an Erlang simulation shows that it works according to the design specification: with the increasing number of social data passing through the network, the nodes learn to filter out irrelevant data, while serendipitous important data is able to pass through the network.

ACKNOWLEDGMENTS

First of all, I would like express my sincere gratitude to my research supervisor, Dr. Julita Vassileva, for providing me the opportunity of M.Sc. study and financial support. Her expertise, understanding, and patience along with her support and guidance have helped me complete this M.Sc. thesis. She has been a great mentor giving me insightful directions whenever I was lost while exploring the research areas. I would like to thank Dr. Ralph Deters, Dr. Chanchal Roy, and my external examiner Dr. Rajesh Karki for being a part of my thesis committee and providing me with valuable advices and suggestions. I want to express my thanks to Jan Thompson and Gwen Lancaster, and other staffs at the department of Computer Science, University of Saskatchewan, for being very helpful throughout my study.

I would like to appreciate my parents Urmila Tandukar and Dhurba Kaji Tandukar, for their love and believing in me. Finally, I would like to extend my warmest appreciation to my love Rajeena Shrestha, for her unconditional love, constant support and endless encouragement. I dedicate this thesis to her.

TABLE OF CONTENTS

PERMISSION TO USE i

ABSTRACT ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

LIST OF ABBREVIATIONS ix

INTRODUCTION 1

1.1 Thesis Outline 4

BACKGROUND AND MOTIVATION 5

2.1 Online Social Network 6
2.2 Decentralizing Online Social Networks 7
2.2.1 Users Hosting Their Own Social Data on Trusted Servers 7
2.2.2 Using P2P Infrastructure 8
2.3 Information Filtering 10
2.4 Information Dissemination 11
2.5 User Modeling 13
2.6 Filter Bubble 15
2.7 Summary 16

PROPOSED ARCHITECTURE AND APPROACHES 18

3.1 System Architecture 18
3.2 Selective Propagation of Social Data 24
3.2.1 Social Network 24
3.2.2 Relationship Modeling 25
3.2.3 Ensuring Serendipity 33
3.3 Summary 34

DESIGN AND IMPLEMENTION OF THE SIMULATION 35

4.1 Agent-Based Modelling and Simulation Approaches 35
4.2 Simulation Design 39
4.2.1 An Agent’s Behaviour 40
4.2.2 Social Network Dataset Used in the Simulation 43
4.3 Implementation Details 44
4.3.1 Parsing and Preparation of the Subset Graph 44

4.3.2 Distribution of Interest ... 45
4.3.3 Simulation ... 45
4.3.4 Analysis Phase ... 47
4.4 Summary ... 48

EXPERIMENTS AND EVALUATIONS ... 49

5.1 Social Graph ... 49
5.2 Parameters ... 50
5.3 Experiment Goals ... 52
5.4 Experiment Setups ... 53
5.5 Experiments ... 54
5.5.1 Experiment 1: Does Relationship Model Filter Out Irrelevant Social Updates? ... 54
5.5.2 Experiment 2: Does Learning Rate Affect the Filtering Rate? ... 62
5.5.3 Experiment 3: How to Ensure Serendipity through Global Popularity? ... 64
5.5.4 Experiment 4: How to Ensure Serendipity through Local Popularity? ... 68
5.6 Discussion ... 70

CONCLUSION ... 73

6.1 Summary ... 73
6.2 Contributions ... 74
6.3 Future Work ... 75
6.3.1 Implementing the Mechanism in a Real Decentralized OSN ... 75
6.3.2 Extending the Approach ... 75
6.3.3 Simulation Extensions ... 76

LIST OF REFERENCES ... 78

DATASET PARSER AND SUBSET CREATOR WRITTEN IN PYTHON ... 83

SIMULATOR WRITTEN IN ERLANG ... 87

LIST OF TABLES

Table **page**

Table 3.1: Categorization of feedback 27

Table 5.1: Machines specification for experiments 53

Table 5.2: Properties of selected subsets of social graph....................53

Table 5.3: Total flow of irrelevant, relevant, and filtered messages....................62

LIST OF FIGURES

Figure **page**

Figure 3.1: P2P architecture..19

Figure 3.2: Users hosting their social data on remote machines 20

Figure 3.3: System architecture 23

Figure 3.4: Relationship model..28

Figure 3.5: Filtering by relationship model 31

Figure 4.1: A community of agents in the simulator 40

Figure 4.2: Algorithm/behaviour of an agent 41

Figure 4.3: Process creation in Erlang...45

Figure 4.4: Process implementation in Erlang...46

Figure 4.5: Erlang slave machine creation...47

Figure 5.1: Subset from StudiVZ dataset (visualized using Pajek [59]) 50

Figure 5.2: Result of Experiment 1 for subsets of 427 nodes..55

Figure 5.3: Result of Experiment 1 for subsets of 1300 nodes..56

Figure 5.4: Result of Experiment 1 for subsets of 2613 nodes..57

Figure 5.5: Result of Experiment 1 for subsets of 5324 nodes..58

Figure 5.6: Result of Experiment 1 for subsets of 7062 nodes..59

Figure 5.7: Result of Experiment 1 for subsets of 9094 nodes..60

Figure 5.8: Flow of relevant vs. irrelevant messages through the network for
learning rate 0.6 63

Figure 5.9: Result of Experiment 2 for subsets of 427 nodes..65

Figure 5.10: Result of Experiment 2 for subsets of 1300 nodes......................................65

Figure 5.11: Result of Experiment 2 for subsets of 2613 nodes......................................66

Figure 5.12: Result of Experiment 2 for subsets of 5324 nodes......................................66

Figure 5.13: Result of Experiment 2 for subsets of 7062 nodes......................................67

Figure 5.14: Result of Experiment 2 for subsets of 9094 nodes......................................67

Figure 5.15: Result of Experiment 3 for subsets of 427 nodes..69

Figure 5.16: Result of Experiment 3 for subsets of 1300 nodes......................................69

LIST OF ABBREVIATIONS

ABM	Agent-Based Modeling
CSV	Comma-Separated Value
FOAF	Friend-Of-A-Friend
LSA	Latent Semantic Analysis
OSN	Online Social Network
RDF	Resource Description Framework
TF-IDF	Term Frequency-Inverse Document Frequency
TSV	Tab-Separated Value
UML	Unified Modeling Language
URI	Universal Resource Indicator

CHAPTER 1
INTRODUCTION

The information age with the rise of internet has brought many changes in the lives of human beings. It has given us the ability to produce and consume information at the rate which would not have been possible otherwise. The way of generating and consuming this information has changed a lot with an introduction of Online Social Networks (OSNs) such as Facebook, Twitter, Foursquare, LinkedIn, Google+, and many others. It has provided a common ground where people are generating and consuming large amount of information. This information varies from personal thoughts (like status updates, photos, etc.) to global news (such as natural disasters, wars, etc.). There are more than 200 popular OSNs available to the users to choose, as listed in Wikipedia [46]. With this explosion of OSNs a given user's data is often scattered over different OSNs. OSN’s streams/timelines, where users get their updates/feeds from friends, has a rapid flow of information, of which only a fraction is relevant to each user. Therefore, recommending relevant information and filtering out irrelevant information have become a research priority.

The currently popular OSNs are centralized, which means they store all the information that people generate and consume in one logically centralized location owned by OSN providers. People have to share their information (status, updates, photos, etc.) with the site, in order to share it with their friends. For example, pictures shared in Facebook will be uploaded on the server and will be hosted there. In this way, users lose control over their own data and it becomes property of the OSN providers. Users’ data is also scattered and duplicated over the internet in different OSN providers which usually do not support data interoperability (apart from trivial user profile information). This is why OSNs are being viewed as information silos [48].

This issue of control over user data and privacy in centralized OSNs has recently motivated research into decentralized OSNs, where the data of users is kept logically in their own clients (nodes). Decentralized and open OSNs have been proposed as a future alternative of current centralized, closed and corporate-owned OSNs by leading web thinkers including Tim Berners-Lee [48]. Such networks allow users to retain control over their data, rather than surrendering it to a corporation. A decentralized OSN can grow organically with the number of users who join, starting from a small cluster of friends, serving, for example, the employees of an organizational unit, a class of students, a group of volunteers, or a neighbourhood. Such small-scale OSNs are of particular interest to any organization that attempts to provide a forum and community to share experiences, expertise, but not surrender all the communication data to a commercial company such as LinkedIn, Google or Facebook. Decentralized OSNs scale well, just as peer-to-peer systems do, without the need of investing in advance in a powerful server and databases that handles all requests and stores all the data (which may not be necessary, if the community flops). Even better, a decentralized social network can survive at a small scale, and does not need to compete with Facebook for world dominance. With a few committed “hub” users, a decentralized OSN that allows for interoperability would have sufficient material to feed a fairly large social network of users.

While there are already some projects (e.g. Diaspora, PeerSoN), aiming to develop a decentralized OSN alternative for Facebook, there is still relatively little research in this area and there are a lot of interesting issues to explore. Among several approaches to build decentralized OSNs, peer-to-peer (P2P) infrastructure is one option. P2P technologies have been popular for building file-sharing applications, but have not yet been explored in the context of OSN. The inherent nature of how people connect with each other in a social network makes peer-to-peer architectures a candidate platform for building decentralized OSNs.

Decentralized OSNs are expected to share some of the same problems with centralized OSNs, for example, dealing with the cognitive overload caused by too much information flow (called social network overload), which demands finding ways to filter out irrelevant data or to empower the user to do that. Filtering information, however, gives rise to the personalization bubble problem [27], in both centralized and decentralized OSNs. The decentralization also poses some specific problems. For example, certain filtering approaches (e.g. collaborative filtering), are impossible in a decentralized OSN, due to the unavailability of centrally stored user ratings, or traces. On the other side, decentralization allows for preserving privacy in user modeling by developing detailed user models stored locally and shared on demand for a given purpose [21]. The decentralization of OSN leads also to some specific problems faced by other decentralized and P2P systems, for example, the need to create a virtual "co-presence" of users, in order to maintain the connections among users and the flow of information. Another such problem is the need to delegate some of the maintenance of the social network tasks, which is done by the central server in centralized OSNs to the individual nodes (clients). For example tasks such as the modeling of users' interests, the filtering of irrelevant data, the identification and protection from spammers, ensuring that important social updates (e.g. status updates, photos, links, videos, etc.) are propagated through the network, ensuring that the users are connected, have to be carried out by the nodes.

This thesis addresses following problems:

1. Privacy and data ownership problems in centralized OSNs.

2. Information overload in OSNs.
3. The filter bubble problem that arises due to information filtering.

The goal of this research is to propose an information filtering mechanism, that is suitable for decentralized OSNs, and allows each node to filter out irrelevant social data, while ensuring a level of serendipity (serendipitous are social data which are unexpected since they do not belong to the areas of interest of the user but are nevertheless desirable because they are popular).

1.1 Thesis Outline

The remainder of the thesis is structured as follows. Literature survey along with the research problem is presented in Chapter 2. Chapter 3 discusses the architecture for decentralized OSNs and approaches on filtering out irrelevant social data while ensuring serendipity. The simulation design and several setups of experiments to evaluate the approach are described in Chapter 4. The results and discussion are presented in Chapter 5. Chapter 6 summarizes the contributions of the thesis and outlines directions for future work.

CHAPTER 2
BACKGROUND AND MOTIVATION

OSNs have provided mediums for people to communicate, share, and consume information (social data) which varies from global news to personal status update. The social network in OSNs consists of interconnected nodes representing users through arcs representing relationships. When shared to the network the social data propagates to every user, who is connected with the sender, whether it is relevant to the user or not. A sender shares only one social data at a given moment. But from the viewpoint of a receiver, she has to view social data contributed by many senders. According to statistics from Facebook [6], the average user has 190 friends. If all of the friends share social data updates daily, then an average user gets 190 different social data updates daily. In this way, the user is flooded with social data, most of which are irrelevant to the user's interest. This thesis propose an approach that provides a way to reduce the propagation of irrelevant social data to users from their friends in decentralized OSNs and ensures serendipity by receiving important/popular social data.

This chapter presents an overview of existing work in the areas of online social networks, decentralized online social networks, information filtering and dissemination, user modeling for filtering mechanism, and filter bubble problem and the need for ensuring serendipity in recommender systems.

2.1 Online Social Network

An Online Social Network is defined as a web platform in which a person can create a profile, connect to other people, view and traverse network of connections, share resources and information within the system, and use social applications with which people within the system can interact and collaborate with each other [11], [18]. With the growth of internet usage, OSN service providers (such as Friendster, MySpace, Last.FM, Hi5, Twitter, Facebook, etc.) have proliferated, and some of them have grown to have millions of active users. All these OSNs follow a centralized, client-server architecture. This architecture supports high accessibility since users can access the service from any web-browser, on different devices, wherever and whenever they desire. But due to their centralized nature, these OSN have weaknesses, such as a single point of failure, scalability issues, as well as privacy and release of control issues from a user's point of view. Central administration is required to manage the activities of users, which gives rise to privacy issues due to the centralized data storage. The centralized management requires larger servers, storage and bandwidth, and a lot of flexibility to accommodate the growing number of users. Partially due to these issues some popular OSN services (e.g. SixDegree.com and Friendster [11]) were discontinued, and their users lost all their social contacts and data. Some centralized OSN services like Twitter, due to the growth in their popularity were having many performance scalability issues and still encounter frequent periods of slow response or even unresponsiveness [17]. In addition to these technical issues, there are also social issues arising with the rapidly growing popularity of social networking. People are becoming more conscious about the information that they share in their social networks. Services like Facebook, which have millions of users, have to deal frequently with inquiries about how they protect their users' privacy. For example, Facebook's Beacon online ad system was tracking activities of the users in third party websites even when users were logged off from Facebook and when they had declined to broadcast their activities [29]. The system caused an outrage among Facebook users, and Facebook quickly discontinued Beacon. Yet, Facebook keeps huge amounts of very private user data, which is company property, and is mined for various purposes, most prominently, personalized advertisement. Present centralized OSN have full control over user data. Once it is shared the user loses control over it and cannot remove it or export it into another OSN. This has lead OSNs being viewed as “information silos” [48]. Instead of this centralized architecture, a decentralized architecture can be considered as an alternative for OSNs.

2.2 Decentralizing Online Social Networks

Decentralized online social networks have a distributed computing structure using a trusted network of servers or a peer-to-peer (P2P) network. In [48], the authors suggest that decentralized OSN will give back to users the control of their data with respect to privacy, data ownership and information dissemination. There exist two types of decentralized OSN. In the first one the users host their own data on their own machines or on servers (cloud) trusted by them. The second way is by using a peer-to-peer architecture. These two types of decentralized OSN will be discussed in the next sections.

2.2.1 Users Hosting Their Own Social Data on Trusted Servers

According to [48], in decentralized OSN the user is not required to take part of social networking services, such as Facebook, Twitter, etc. to maintain his/her online social presence. The user can host a FOAF[1] [12] (Friend-Of-A-Friend) file, an activity log, photos/videos, and social client on a trusted server. The user has full control over what social data to share and with whom. The authors [48] describe how the functionality offered by popular social applications,

[1] http://xmlns.com/foaf/spec/

such as “Personal Wall”, “Photos”, and “News Feed” can be implemented in decentralized OSN. They propose a system in which the user shares and communicates social data with other users by using WebDAV [45] or SPARQL Update[2] [38] protocols. As a prototype a system was developed called “Tabulator” [7] which is a generic data browser and editor of linked RDF (Resource Description Framework) data [8]. This decentralized OSN encourage users to store their social data on the web in standard format such as RDF and it should be accessible through URI (Universal Resource Indicator). Therefore, the user does not have to rely on only one social application, and can use any social application that supports these open technologies.

2.2.2 Using P2P Infrastructure

Decentralized OSN can also be implemented using a P2P network. A P2P network is a distributed network in which nodes are connected with each other to participate in processing, memory, and bandwidth intensive tasks. These networks scale better than centralized server architectures without the need of costly centralized resources. P2P networks have been popular mostly as file sharing networks (such as KaZaA, BitTorrent, etc.) and as VOIP collaborative networks (such as Skype), but have not been used as a medium for online social networking. The inherent nature of peer-to-peer connection between users in a social network makes OSN a good candidate for peer-to-peer architecture [18]. In file-sharing P2P systems like Gnutella, most of the users are free riders [2]. These users tend to consume more and contribute less to the network. In contrast, P2P applications such as Skype have more stable network since there is an incentive for the users to be online. Users of Skype stays connected to the network so that they can communicate with their friends and receive calls; as a result of being online, their servants are active, route P2P traffic, and keep the network stable. Skype is a widely used peer-to-peer

[2] http://www.w3.org/Submission/SPARQL-Update/

VoIP client and has around 15 million concurrent users [14] (at the time of writing, this number has grown to 27 million). Skype can be considered as a social application that helps in connecting two or more users to share information through voice and Instant Messaging. The popularity and scale of Skype shows the potential that P2P holds as an implementation infrastructure for OSN.

To accommodate the familiar functionality of centralized OSN (status updates, photo uploads, commenting, rating) in decentralized OSN, one has to overcome various challenges. Since the social data is stored at the peers, the availability of the social data depends on the online behaviour of peers. Storing data on the peers allows encryption that can ensure privacy while transmitting data from peer to another. The propagation of social data or updates among users in the OSN should be managed so that there is less duplication and no latency. These and other challenges have been discussed in detail in [18].

Some P2P systems have exploited properties of social networks (such as trust and collaboration) in other to improve the performance of P2P networks. Some systems like Tribler [30], have adopted social networking on a BitTorrent based P2P file-sharing network in order to recommend, search, and download contents. In [30], the authors have developed a “Buddycast Algorithm” that exchanges preferences of peers in the implicitly defined social network to generate a recommendation list and search contents. By using a collaborative download protocol called “2Fast”, in which users collaborate to contribute their bandwidth, download performance was also improved. In [3], the authors have even used the graph topology of real social networks in order to form a P2P overlay topology and used it to improve lookups and scalability.

PeerSoN [13] is an effort to build decentralized OSN over a P2P architecture. It implemented encryption of user data to protect privacy and ensure direct exchange of data between devices for delay-tolerance and opportunistic networking. It uses OpenDHT [35], a Distributed Hash Table (DHT) service, for look-up of other peers in the P2P network and also to store data, such as, IP address, file information, and notifications for peers. In DHT [32], {key, value} pairs are stored in distributed nodes and a node can retrieve the value associated with any key efficiently. The prototype of PeerSoN provides functionality to create social links (becoming friends), storage to maintain profile and content of post by their friends, asynchronous messaging, and also live chatting.

2.3 Information Filtering

Since in OSN users consume information passively by watching the stream of status updates posted by their friends, the problem of dealing with information overload in OSN could be considered as an information dissemination problem, where the user assumes a passive role, while information is routed appropriately to the user. OSN produce large amount of information and the propagation of this information to its destination has to be well coordinated so as to reduce overload, duplication, latency and to ensure quality.

The reduction of the information overload can be achieved by filtering out the irrelevant information. Filtering of irrelevant information and providing users with personalized recommended information are problems addressed by recommender systems. Recommender systems adapt to the needs of specific individual users and deliver the most relevant information for their needs [34]. Generally recommender systems follow two main approaches: content-based or collaborative filtering. In content-based recommenders the information is filtered according to a user model developed by analyzing the information previously used or liked by the user. In collaborative recommenders information is filtered to the user according to what other users with similar previous usage or ratings have liked [33].

Recommender systems can be exploited to identify information about any particular user [31]. Ramakrishnan et al. [31] has described the risk of deducing connection between users and combining connected information to identify personal details of the user. This is a realistic possibility since the data necessary for recommendation are typically hosted in centralized servers. Companies that offer personalization services, often provide their database to third-party consultants for statistical analysis. Narayanan et al. [25] were able to get sensitive personal details from the anonymized dataset provided by the popular movie recommender system Netflix by matching it with large number of personal movie ratings available from a popular online movie database IMDb[3] (Internet Movie Database) [49]. The privacy breach can be achieved using the experimental dataset provided by the site, containing a personal data of the users which may include sensitive information. One way to counter this issue is to use decentralized architecture where users data are not stored on a company owned central database or centralized architecture but on a trusted server where the user wants to keep her data, and make available only through permissions by the user. The following section describes recommendation mechanisms that are suited for decentralized systems.

2.4 Information Dissemination

Information dissemination approaches provide an alternative to recommender systems in decentralized systems, where instead of personalized filtering of information, information is propagated selectively through a social network following various models, such as infection models used in studying how diseases spread in a population, or how innovation spreads in communities. The Push-poll recommender algorithm [43] propagates information through an implicit social network, formed by peers with similar interests. It takes into account feedback

[3] http://www.imdb.com/interfaces

from the recipient to determine the future influence of sender on recipient. The KeepUp recommender system [44] is based on the Push-poll algorithm. It allows the user to interactively adjust the amount of influence that her neighbours have on the recommendations she receives. This gives power to the user to decide indirectly what and how much information is propagated to her. GoDisco [19] focuses on dissemination of social data according to the context of the information. The nodes (users) gossip with their neighbours periodically about the strength of their interests. They also keep track of the behaviour of their neighbours (like activeness, forwarding behaviour). This knowledge of each other is used in the dissemination phase where the messages are assumed to have some semantic value that can be mapped to the interests of the nodes.

In a file-sharing P2P networks, users engage in proactive search by sending queries about resources of interest and the system returns lists of their locations (i.e. peers that store these resources). However, also in the area of P2P, Koubarakis et.al [23] have proposed information dissemination approaches – a “selective information push”, where the user posts her profile to “super-peers” and receives notifications about resources that match her interests as these resources become available. Since the user is both the consumer and producer of the resources shared in the network, she can also post advertisements of her resources and the super-peer will push notifications about these resources to the relevant peers (peers with matching interests). This mechanism depends on the preferences of the user, querying super-peers and if the user has a more general area of interest, she might get too many notifications. Here, the super-peer can be considered as a recommender that is pushing information according to the peer’s interest.

Sun et al. [39] have applied user modeling and modeling of relationships between users. With the help of user models of interest, the nodes are able to route queries to other users with similar interests. By modeling relationships between users, the nodes can determine typical time patterns of neighbour's behaviours to route requests appropriately and ensure better quality of service. Thus, through the relationship models, a virtual overlay topology over the P2P network is created. A relationship between users is created when a user successfully downloads a file from another user and the strength of the relationship grows with the number of successful interactions between these users. However, whether the strength of relationship between users should help to propagate social data depends on the area of interest. Two people in a strong relationship (e.g. siblings) may have very different tastes in music or different political views. If one of these users has liked a particular information item, it does not mean that the other one would like to see that item too. Therefore, relationship models need to take into account the users interests. Modelling user interests has been studied by the area of User Modeling.

2.5 User Modeling

Each user has her own characteristics, e.g. interests, preferences, etc., and a model of these characteristics can be utilized to recommend her relevant information. These characteristics comprise the user model for the system. User models as discussed in [37] can be classified into canonical vs. individual models, explicit vs. implicit models, and long-term vs. short-term models. Canonical is a model that tries to incorporate the average characteristics of user. As a user population contains different individuals that differ from each other, systems should incorporate individual user models. Explicit models require the user to explicitly define her characteristics, while implicit models are updated automatically by the system, according to the user's interaction with the system. Explicit models provide a way to overcome the cold start problem that can arise in implicit models, since at the beginning (when the user has not interacted with the system sufficiently) the system knows nothing about the user. Short-term and long-term models can be distinguished from each other based on the amount and nature of data about the user that is collected and the duration for which it is kept. The user model is updated using feedback from the user's interaction with the system.

User models have been central part of recommender systems, especially content-based recommenders that rely in representations of user interests, to generate personalized recommendations. The quality of recommendation depends on the user characteristics represented by user model and also by the accuracy of the model, volume of information in the model, and so on. User models are used in many recommender systems which offer recommendation for movies, music, news, books, research papers, TV programs, and many more. One of the first recommender systems, using and explicit user model was Grundy [36] which suggests relevant books to the user according to their interest with the help of user model built on the basis of explicit interaction with the user, inferring to the user's actions and referring to stereotypes. Stereotypes are used in Grundy to tackle the cold-start problem which plagues all recommender systems. Instead of stereotypes, WebMate [16] uses TF-IDF (Term Frequency-Inverse Document Frequency) method to identify fingerprints of documents seen and liked by the user and build a user model consisting of terms characteristic for documents that were liked by the user. WebMate provides relevant similar web documents to the document fingerprints seen by the users based on vector space model, where each vector space consists of a word and the number of times that word occurs in the document. Using feedback from users for the creation of user's preferences and interests has been common for many systems that provide relevant information to the users. NewsDude [9] is a news agent that uses classification approach to recommend relevant news articles to the user by inferring to user models which model user's short-term and long-term interests: short-term model is used to recommend similar articles using the nearest neighbour algorithm, long-term model is used to predict interests in articles which cannot be classified by short-term model, using a Naïve Bayes approach to classify news articles as relevant or irrelevant.

2.6 Filter Bubble

As discussed in earlier sections, in the process of filtering out irrelevant information, there is a danger that the system adapts too well to the user's preferences and therefore, the user will miss unexpected but desirable information. The personalization algorithm has helped users find relevant information according to their areas of interest but it has also confined them within their preferred personalized space. This personalized space of the user where she is getting information that belongs to her preferences only is referred to as "filter bubble" by Eli Pariser [27]. The existence of such filter bubble contradicts the users' need for variety in information, discovery of new information or interest and occasional change in their interests. For example, person who is not interested in sports will not be informed about the Olympics even if a person from her hometown won a medal! Sometimes, surprising discovery of new information might result in creating new interest in a person. Therefore, we should not neglect the "filter bubble" effect which prevents these surprising discoveries.

In [22], the authors defined such surprising discoveries of information as "serendipity recommendation". Serendipity can be confused with novelty and the authors take care to explain the difference between novelty and serendipity: novelty as information recommended which could be discovered by the user somehow; and serendipity as information recommended which could not have been discovered by the user if the system did not recommend it. In order to implement serendipity computation, the authors have used poor similarity index strategy in their content-based recommender system. This strategy includes a heuristic which takes into consideration that the documents which are dissimilar to previously liked items by users are mostly serendipitous ones.

The usefulness of items recommended by recommender systems cannot be guaranteed on the basis of accuracy alone. Hence, serendipity is considered in the recommender systems to increase the user satisfaction by recommending items which are unexpected from the traditional prediction systems [24]. In [24], authors have proposed metrics to evaluate the serendipity in recommender systems by measuring unexpectedness. Unexpectedness has been defined by them as a measure that indicates how deviated a result is from traditional recommender mechanism.

2.7 Summary

With the large amount of information flowing through OSN tied to user's personal interests and data, there should be serious consideration for user data privacy and ownership of data. Decentralizing the online social network and giving users control their social data themselves provides a way to ensure privacy and ownership for users over their data.

With the growing size of an online social network and the growing amount of data shared, information overload for users becomes inevitable. Recommending relevant information to the users can help the user focus their attention to important information rather than on the clutter of unwanted information. Recommender systems using various mechanisms for filtering relevant data for the user have been proposed, but these systems are mostly designed to work with centralized architecture. There has been some work on decentralized information filtering using information dissemination and approaches using modeling of relationships between users, which seem promising.

Though filtering techniques reduce information overload, they can encapsulate the user in filter bubbles in which they receive only the information of their interest. This will isolate users from getting information which could be important and useful, but does not belong to their areas of demonstrated interest. Introducing some serendipity in recommender and filtering mechanisms can help the users discover new information and new areas of interest.

We propose a decentralized online social network to give users control over their information. To deal with information overload in decentralized online social network we propose an approach which builds relationship models for friends overlaid over different areas of interest. These relationship models forms a base of our filtering mechanism to ensure flow of relevant information for each user in the network. To decrease of effect of filter bubble due to our filtering mechanism, we propose introducing a mechanism based on the popularity of information, which will ensure a flow of serendipitous information in the network.

CHAPTER 3
PROPOSED ARCHITECTURE AND APPROACHES

This chapter will present the approach to tackle the issues of privacy and information overload in social networks. We propose a decentralized architecture to give users more control over their social data. Interest-based modeling is done by each node (user) to filter out irrelevant information, while ensuring the flow of serendipitous information. Section 3.1 presents the system architecture for decentralized OSNs. Section 3.2 describes how to filter out irrelevant information from the network while maintaining serendipitous information.

3.1 System Architecture

As explained in chapter 2, OSNs can have decentralized architecture either by:

1. Using a Peer-to-Peer (P2P) architecture where user data is hosted on their p2p nodes (clients).
2. Hosting the user data on remote machines (at securely accessible locations).

To accommodate familiar functionalities of OSN (like status updates, photo uploads, commenting, rating) in a decentralized architecture, especially in a P2P architecture, there are various challenges [18]. The propagation of social data among users in the P2P OSN should be managed so that there is less duplication and no latency. Using P2P architecture, data are stored at the peer nodes and the availability of the social data depends on the online behaviour of peers. So if the node is inactive (i.e. the user's machine is offline) then the social data associated with that users will be unavailable to her friends. Figure 3.1 shows a typical P2P network, in which machines are connected with each other over the internet and the peer identity depends on its IP address. This leads to a complication: when the machine on which the peer node resides is physically moved, its IP address is changed, and it becomes difficult to discover that peer and link it to the network. With the proliferation of mobile devices, such as laptops, tablets and smart phones, users are moving their machines between networks frequently, and this problem becomes significant.

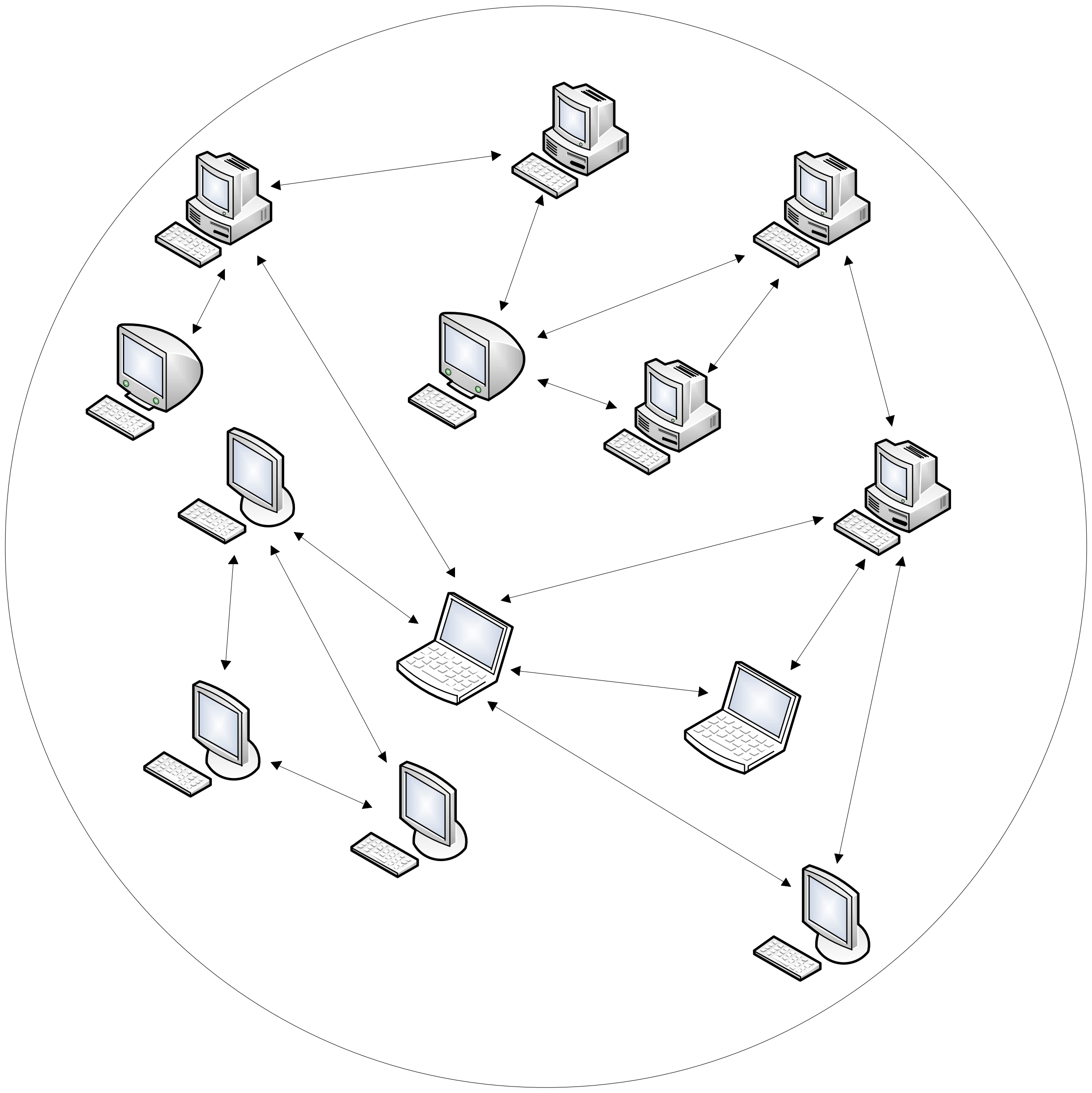


Figure 3.1: P2P architecture

In contrast, if users store their social data in some standard format such as RDF on the web in a trusted server or cloud, which can be accessed through URI (to ensure presence in the network), they still maintain control over their data, and the data becomes accessible from the various devices belonging to the same user. This also makes the data readily available all the time and it does not depend on the uptime of the machine where the social data resides. Figure 3.2 shows a network of users which store their social data on trusted remote servers/clouds.

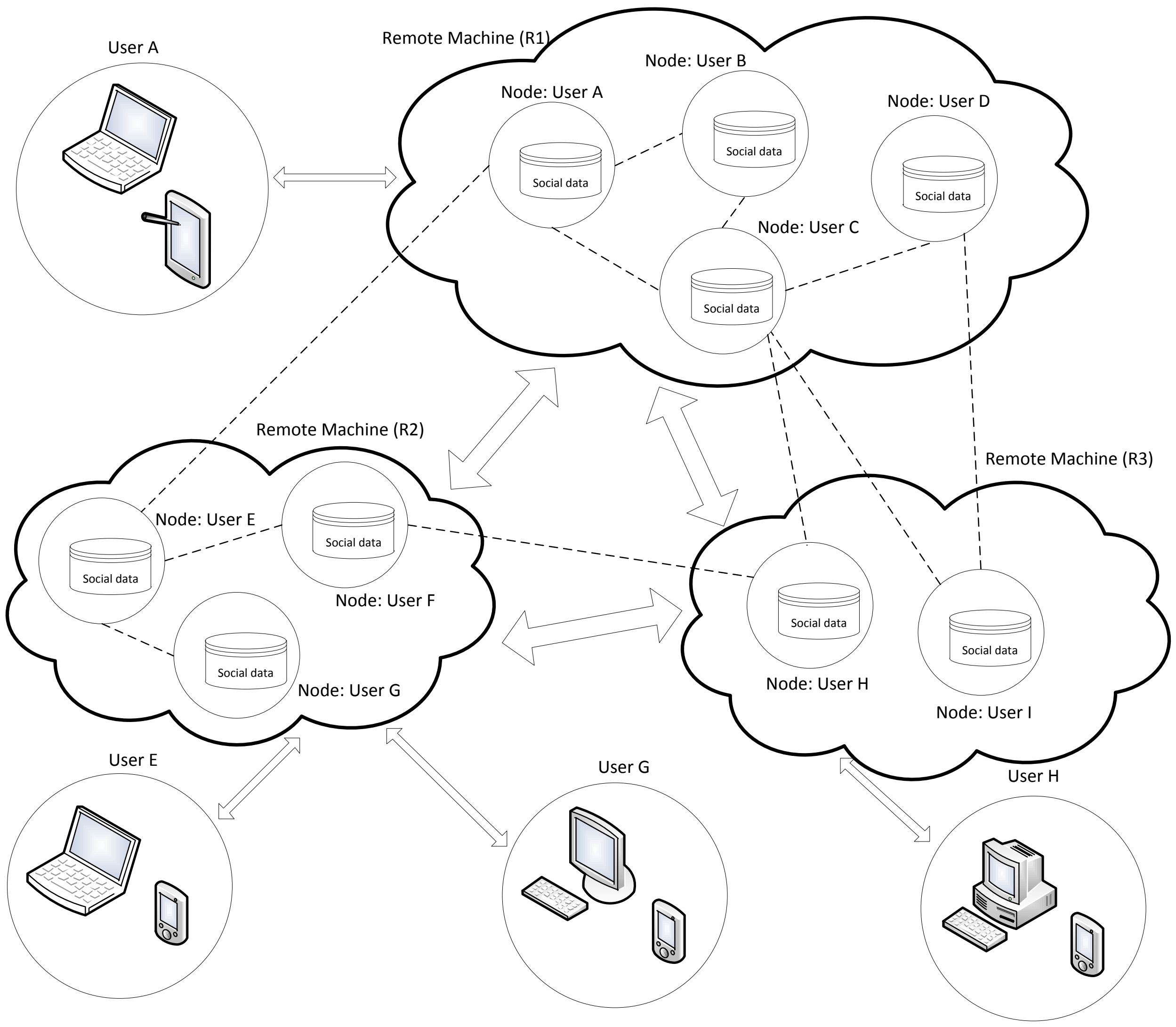


Figure 3.2: Users hosting their social data on remote machines

These remote machines are servers which can be accessed through network (typically internet). The hardware and system software which provides the computing services over the internet are referred to as a cloud [4]. Therefore, we can refer the above remote machines in Figure 3.2 as clouds. As shown in Figure 3.2, each cloud can host social data and application of many different users. A dotted line in the figure represents a social link (friendship/colleagues/common interest) between users. The link can also extend from one remote machine to another and the remote machines can interact with each other to send data updates between each other. When the social data and application is hosted on cloud, the user can access her social data and application using any device via the internet. The transfer of information is done via social links within or across these remote machines and the availability of social data is better than in P2P infrastructure.

That is why a decentralized architecture is chosen in which the client is storing the user data on a cloud accessible via the internet. For security it can have a secure connection too. The user data includes the social updates, the relationship model of the user, and an expandable list of interest areas. When a node (user’s social application and data as a whole) forwards a social update to another node (of a friend user), it consists of the URI of the social update and an access permission i.e. a key that the recipient can use to access the actual contents of the update, as stored on the sender user's storage location in the cloud. The access permissions can be shared and forwarded among the nodes, in the same way as posts are currently reposted by users for their friends to see on Facebook. Yet, unless the recipient makes the effort to copy and save locally the content of the social update (Web URL, or text, or image), the data is only stored on the user's secure storage location, always under the control of the user who shared it in the network, and can be easily removed by him or her. While some users may copy the content of their friend’s social updates, it will happen only occasionally since it requires extra effort, and will be in no way comparable to the massive surrendering of social updates that users currently are forced to do by the centralized commercial OSNs.

In our decentralized architecture, as shown in Figure 3.3, each user has database of social updates and relationships stored on the secure location of the user and a social application to interact with the social data (owned and shared by friends). A machine on a cloud can host the social data of more than one user but the data is stored securely and is unavailable to anyone except friends of the user, to whom the user has sent permission to access it, and this is only to the data involving the specific social update that is being sent. A variation of this design would be to have both the user data and application hosted on a cloud, and the user logging into her application from anywhere using HTTP (rather than having copies or different versions of the application running on different devices).

The system can be viewed as a multi-agent system where agents are distributed on various machines on the network. Each agent is a web application which is accessible through URI and has a database to store the user's social data. The user's friends list, their locations, shared information, relationship model and other social data is stored in the local database and managed by the agent. Several agents may reside in one machine and can interact with each other (as shown in Figure 3.3, Remote Machine R1), if their users are connected to each other in their social graph. The user can generate and access her social data through the interface provided by her agent. For simplicity, this interface can be implemented as web pages through which user can communicate with her agent (web application) through the http protocol.

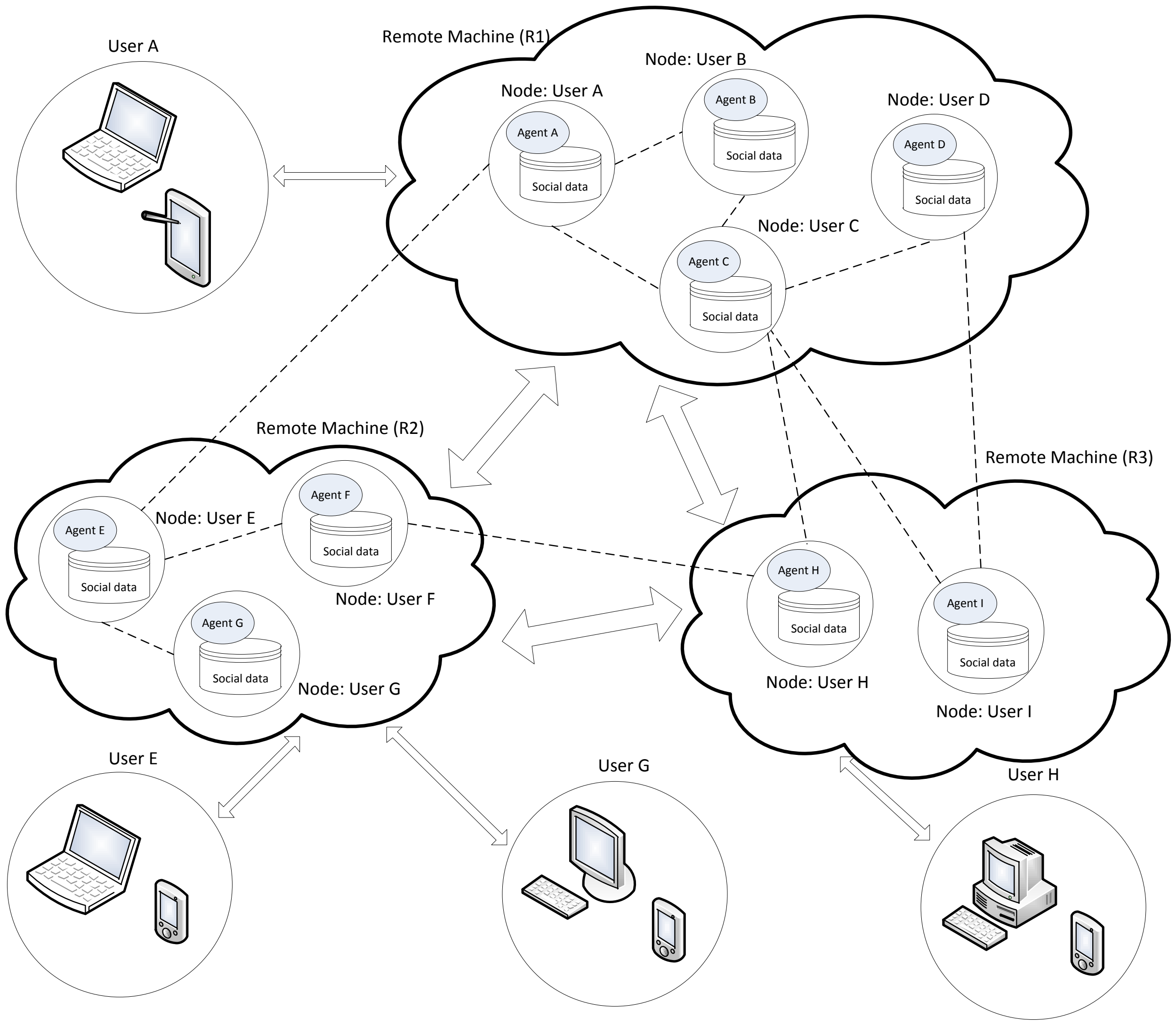


Figure 3.3: System architecture

The agent can be considered as a representative of the user. The user shares social updates first to her agent and then the agent forwards the updates to the agents of her friends. A URI and security key is sent to friends with whom the user wants to share, instead of the social data itself, which allows the friends to view these data from the user's database. The agents take responsibility of analyzing the incoming social data so as to determine whether to display the data to the user of not (filter it away); it also calculates a feedback value for each incoming social data in order to update the relationship strength between the sender and the user so as to be able to filter irrelevant information appropriately later on. The approach of selective propagation of social data with the mechanism of relationship modeling of friends in social network is discussed in the next section.

3.2 Selective Propagation of Social Data

An approach of selective propagation of social data (i.e., information shared by users in social networks, such as status updates, photos, links) by modeling the interests of friends is proposed next, that ensures that social data reaches only the relevant users for whom it would be interesting.

3.2.1 Social Network

The social network can be represented using a social graph. Social graph is a graph in which each user is a node and relationships between users are the edges. Let G be a social graph represented by $\{N, E\}$ where $n_i \in N$ represents the set of nodes (users) and $\{n_i, n_j\} \in E$ represents the set of edges (relationships) between nodes. We can say $n_a \in N$ has some relationship with $n_b \in N$ *iff* there exists $\{n_a, n_b\}$ *or* $\{n_b, n_a\} \in E$.

Information flows in a social graph following the relationships between the nodes. If we decide to distinguish between different kinds of information flowing in the social graph, depending on different areas of interest, we can define different kinds of edges between nodes, corresponding to different areas of interest. Then we may have several edges connecting the same pair of nodes, representing information flow in several different areas of interest. If we look at only one kind of edges at a time, we will have different graph topologies over the same nodes, representing flows of information for different areas of interest. The main idea of the approach is exactly this: to connect users with different edges depending on the area of interest they tend to exchange most information. When the area of interest for a new social data is known, it can be propagated through the network along the edges representing this area of interest, where they exist, thus filtering out nodes/users that are not connected with edges in this area of interest.

To route relevant social data to users, each user or node in the graph will model the interests of other users with whom she has relationships. From the point of view of a given user, the model of interests of other users is considered as relationship model since it signifies how many positive interactions have happened between the users in the context of particular area of interest. Positive interaction between two users in a given area of interest means that user A has sent to the user B social data related to the area of interest, and the user B has given positive feedback to the user A after receiving the social data. As a result of positive interaction, the strength of the relationship between the two users in the area of interest increases. The relationship model is used by the node (user) to adaptively filter social data related to a given interest area, by displaying the received social data only if the user has sufficiently strong relationship in the interest area (*I)* with that friend.

3.2.2 Relationship Modeling

An interpersonal contextualized (with respect to *I)* relationship model is used as a filtering mechanism for irrelevant information. The relationship model consists of a representation of the relationship strengths between a user and her friends in different areas of interest. The strength of relationship between two users is contextualized according to a particular category of interest. The intuition behind this is that two people can be friends, but not share the same level of interest in different topics or categories of interest and may not trust each other’s judgement with regard to these categories. Therefore, a user may be interested to receive updates from a particular friend about, say fashion, but not about politics or health. On the other side, the same user may be interested to hear about health topics from another friend, yet, she may not be interested in that friend’s updates about fashion. Adding a semantic dimension to a relationship adds complexity of representation and computation, but it allows the flexibility to filter both based on the source of the update, i.e. the friend who sends the update and the semantics of the update (category of interest). The relationship models are stored in the user's trusted servers as shown in the figure illustrating the decentralized architecture of the proposed approach (Figure 3.3).

The strength of a relationship from one user to another in a given category of interest is based on previous interactions related to this category of interest. In general, to determine the area of interest of the shared information, users have to either tag their updates with the interest areas or the system has to classify according to its semantics each shared social update. However, supporting full semantic categorization would be probably unnecessary complex and won’t add much in terms of filtering functionality. That is why it is better to use a certain fixed number (for example, 10 or 20) of predefined very general categories, similar to those used in Yahoo or other news sites, e.g. politics, news, technology, sports, health, fashion, living, art, games, humour, etc.

3.2.2.1 Feedback

The relationship strength between the sender and the recipient user is updated using the feedback that the recipient has provided. The feedback is based on the actions of the recipient and influences differently the relationship strength. Table 3.1 gives the details of the feedback according to the actions the user takes on the incoming social data. One can see that higher feedback ($0 < F < 1$) is reserved for actions showing that the recipient has demonstrated some interest in the social data.

Table 3.1: Categorization of feedback

Type	**Actions**	**Feedback**
Type 1	Comment / Share	0.9
Type 2	Rate / Like	0.7
Type 3	View / Open	0.5
Type 4	Ignored / Not open	0.3

For example, if the social data is viewed and re-shared, this would increase significantly the relationship strength. Viewing and commenting or rating will also increase the strength. Just viewing and not doing anything else would slightly reduce the relationship strength in the category of the social update.

The relationship model depends on the previous interactions between two users, and the priority of a new social data is determined according to the previous history of interaction between the two users. The data structure of the relationship model of a user consists of a list of areas of interest and the corresponding strength of the relationships between the user and her friends. Figure 3.4 illustrates the structure of a relationship model that a user maintains in her data storage. The relationship model has a list of areas of interest and a value showing how much the user is interested in each one of them. It also contains the relationship strength on each of the areas of interest for each of her friends also.

Initially, each user keeps a relationship model with respect to all possible areas of interest for each of her friends on her friends list. Naturally people do not discuss or see social data on all topics from all of their friends, but selectively choose to share and discuss information about certain topics that are of mutual interest. By updating the relationship model, the strength of relationship for certain topics/areas will weaken and ultimately disappear and thus social data by the friend with whom the relationship strength on that topic is low will be filtered away. Thus gradually, over time, the system as a whole through interaction of all the nodes and feedbacks will learn about users' shared interests, and users will only see relevant social data from their friends with whom they have strong relationships.

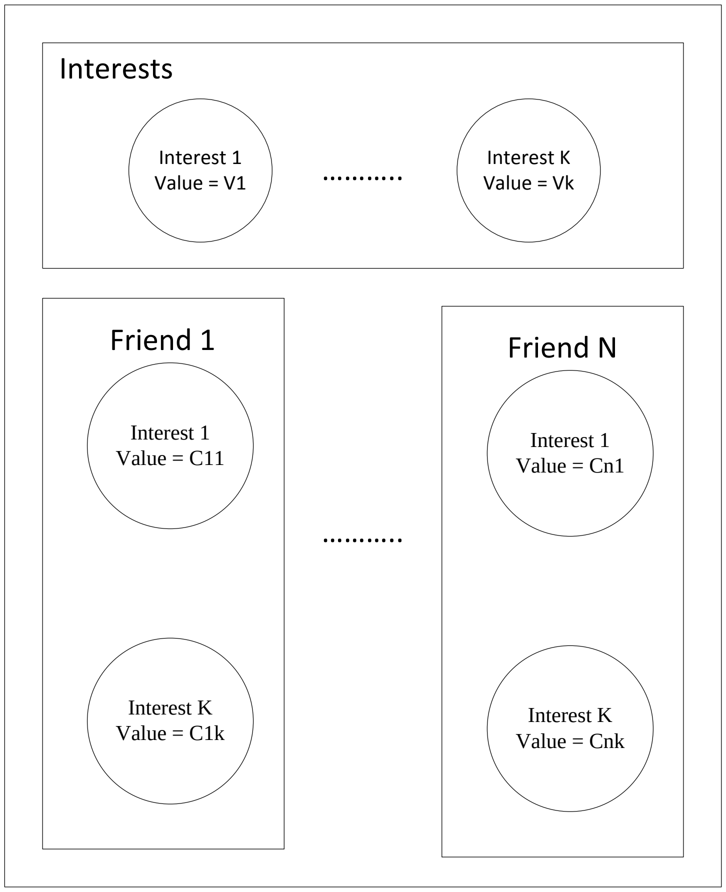


Figure 3.4: Relationship model

3.2.2.2 Relationship strengths

The updating of relationship strengths happen after each feedback from interaction between the users is received. The strength of relationship between users A and B for an interest area I should increase with stronger feedback and decrease with weaker feedback for the social data sent from B to A. The updating of the relationship is calculated using a formula for simulated annealing (reinforcement learning):

$$S_A^B(I) = \alpha * S_A^B(I)_P + (1-\alpha) * F \tag{3.1}$$

Here, $S_A^B(I)$ is the new value of the strength of relationship from A to B, for an interest area I and $S_A^B(I)_P$ is the previous value of the strength of relationship. The parameter $\alpha \in [0,1]$ is a learning rate of the system. Since both User A and User B compute independently their relationship models, they are different, so it is important to note that $S_A^B(I) \neq S_B^A(I)$, i.e. the direction of relationship matters. The feedback that B gives to A for the social data that A sent to B is denoted by F_{BA}, and its value varies from 0.3 to 0.9 as specified in Table 3.1.

The first version of the mechanism presented in [40] assumed that the relationship model is updated by the sender of social update and used to filter outgoing information to her friends. This approach reduced irrelevant information from flowing through the network, but it created a possibility for the model to be tampered by malicious senders to spam other users on the network with irrelevant information. Also, there is no particular incentive for the sender to filter out outgoing information, considering that maintaining the relationships model involves some space and computation costs. Therefore, the approach was modified so that the relationship model of the recipient is updated. Thus the recipient is interested in maintaining the model since it protects it from irrelevant information and from spammers. Instead of sending feedback to the sender so that the sender can update the relationship strength in her model (as in the previous version of the mechanism described in [40]), the recipient updates her model - the relationship strength between the recipient and the sender for the area of interest of the shared social data will be increased or decreased, depending on what the recipient did with the social data [41]. The updated relationship model is used later by the recipient to filter out incoming social data in this category.

If the user doesn’t care for news received from a given friend in a given category of interest, the relationship between him and the friend in this category of interest will fade away and thus the incoming information from the user’s friend that belongs to this category will not be shown to the user. However, social data in other categories will continue to flow freely from this friend to the user. Thus gradually, over time, the system will learn (the knowledge consists of the updated relationship models) and propagate information only along strong relationships, i.e. only information of interest to the users will be received by them. The relationship strength is unidirectional, which means the fading of relationship strength between user A and user B on interest I (i.e. B not reading, commenting, or forwarding social data received from A) does not imply that the relationship strength between user B and user A on interest I will decrease too. It may actually increase, if user A reads and comments or forwards the social data received from B in this category of interest. In Figure 3.5, it can be seen clearly that the social data with category of interest I_1 sent from User B to User A is filtered by relationship model since the relationship strength for User B in User A’s relationship model on category I_1 is very low 0.2 (if 0.4 is taken as threshold value). On the other hand, if social data from User A is sent to User B, it passes through the relationship model since User B’s relationship strength for User A for category I_1 is 0.9. This flow shows that the relationship model is unidirectional, i.e. a strong relationship from B to A in a given area of interest doesn't imply a strong relationship from A to B.

3.2.2.1 Setting of Parameters

While updating relationship strengths using user feedback and equation (3.1) for the learning process, it is very important to maintain the connectedness of the network. One risk of using the reinforcement learning formula is that the system may learn too fast, resulting in lesser connected network. There is a danger that the network with very little connectivity will not carry enough social data to maintain user interest in using the application. Hence, we have to be careful not to reduce the strength of relationships too quickly. Therefore there is a need to update the relationships conservatively. This can be done by setting α in equation (3.1) so as to have more conservative updating.

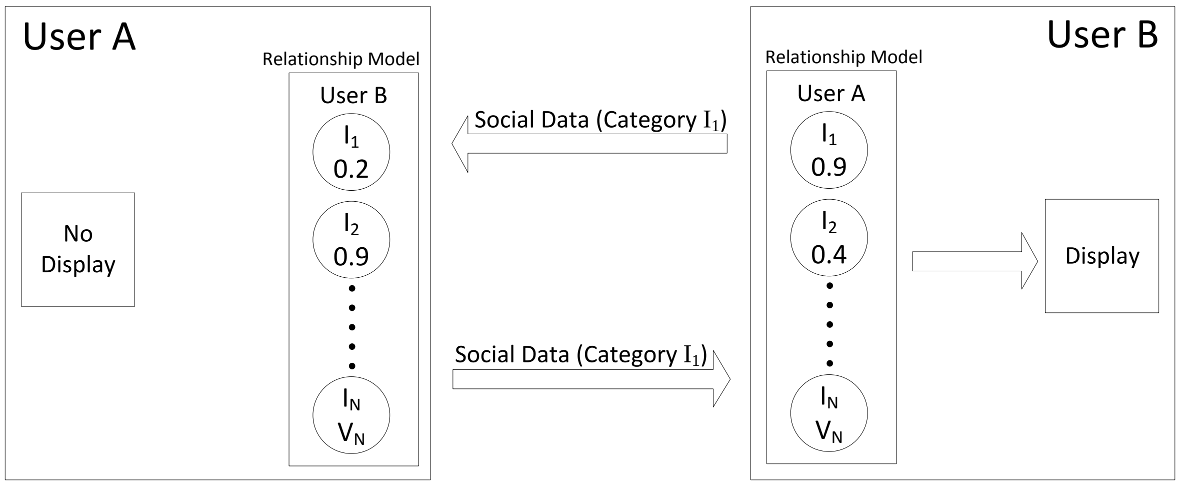


Figure 3.5: Filtering by relationship model

It is taken in consideration that if information is relevant to a user, she will at least open the message and the feedback value is chosen to be 0.5 which is more than the critical value. Through empirical study the critical value for the process of filtering is taken as 0.4 which gives enough interaction between users to update relationship strengths. If the information is irrelevant to the user, she will ignore it and hence the feedback value is chosen to be 0.3, which is less than the critical value and reduces the strength of relationship. Therefore, for the propagation of social data belonging to area of interest *I*, the strength of relationship between the user and her friends should be more than the critical value. Initially, the strengths of relationships for all users with respect to each area of interest among all friends are set to 1; they will decrease and possibly increase again, according to the interactions between the user and her friends.

As the strength of relationship for a particular interest *I* in user A for user B decreases under the critical value, the user A will not get any social data related to interest *I* from user B. For user A to get social data related to interest *I* from user B, she has to make the relationship strength between her and user B stronger than the critical value. To give more control to the users over their relationships, it is also possible to allow users to directly adjust the relationship strength with other users via an appropriate GUI, similar to the interactive influence adjustment deployed in the KeepUp Recommender System [44].

Since the proposed approach is implemented in a decentralized architecture, the computation overhead of maintaining the relationship models is distributed among all the peers. Even in popular OSNs, such as Facebook, an average user has 190 friends [6]. Considering 10 categories, each agent has to maintain 1900 relationships (real numbers) using the reinforcement learning formula, computable in linear time on the number of relationships. So the calculation and manipulation of relationship models for this number of friends and a limited number of categories of interest is not computationally expensive.

In summary, a new content-based filtering approach is proposed that uses a model of users' interest in certain topics that is overlayed over a model of the social relationships of the user. This extends classic content-based filters by allowing filtering information based both on the origin and on the interest level in its category. Using feedback based on the users' actions on the received social data, relationships between users will grow or fade away in context to certain categories of interest adapting the filter to the interests of the individual user, and leading to lesser overloads of irrelevant social data.

3.2.3 Ensuring Serendipity

All information filtering mechanisms suffer from the filter bubble problem [27]. The user becomes gradually isolated from information which could be important but not in her area of interest. *Serendipitous* information is defined as information, which is unexpected, but desirable. Unexpected, because the information does not belongs to user's areas of interest and desirable because the information is popular and important.

In the approach, as described above, like in other information filters, the user will not be able to see serendipitous information coming from friends with whom the relationship in the area of interest of the information has become weak. We need to allow such serendipitous information to reach the user, especially, if it is important information, for example, rated highly or commented by many other users. To allow such information to pass through the filter imposed by the relationship model, an extra parameter is considered for the social updates, in addition to their category of interest – the popularity of the social data. The popularity parameter is a property of the social data and can be calculated as the reputation of the source, or the count of the re-shares or the number of ratings (e.g. likes in Facebook or +1s in Google+). We have categorized the popularity in two types: 1) global popularity – which is like a reputation of the source or number of views that is same throughout the network, and 2) local popularity – which is calculated according to the number of shares, ratings by the user's friends (among connected users). When a social data with a high popularity parameter is encountered, it is propagated without being filtered by relationship models, i.e. the user will see it despite not being interested in the category of the update or in the source of the update.

Since the information belongs to a category that is not of interest for the user, viewing/sharing of this information should not affect the relationship strength between the receiver and sender. The effect of such popular irrelevant information is not considered while updating the relationship strength.

3.3 Summary

Architecture for a Decentralized OSN is proposed, which allows users to either store social data on their desired location or use P2P architecture to host it in their own machines. The architecture includes a client application (agent) with a filtering mechanism based on an interpersonal relationship model. This model contextualizes the relationship strength between two users according to different areas of interest and updates the interest-based relationships using feedback from user actions on the received data. To ensure a flow of serendipitous social data while filtering, a mechanism is proposed in which popular social data are allowed to pass through the filter, based on their reputation/popularity.

CHAPTER 4
DESIGN AND IMPLEMENTION OF THE SIMULATION

Computer simulation is a computer program that is used to model a real world system, and can be run to generate results. These results can be compared with data obtained from the real world, and this can help in validating the model, improving it, and making predictions regarding future behaviour of the system.

This chapter describes the design and implementation of the simulation for the proposed mechanism described in chapter 3. Section 4.1 presents agent-based modelling which is one of the popular ways of performing simulation and tools that facilitates it. Section 4.2 describes the design and implementation details of the simulation system for the proposed mechanism in chapter 3.

4.1 Agent-Based Modelling and Simulation Approaches

One of the popular ways of performing simulation is through agent-based modeling (ABM). In the area of multi-agent systems an agent is defined as an autonomous entity capable of interacting with other agents, responding to environment stimuli and having a goal-oriented behaviour. Environment stimuli can be availability of resources, population growth, change in variables, and many more. Agents in the simulation can have the same or different goals. These agents are generally rule-based and capable of interaction with little or no guidance from a central authority. Due to interacting behaviours of these simple rule-based agents, the emergence of group behaviour pattern can often be observed. Therefore, through agent-based simulation we can study the emergence or changes in group behaviour resulting from changes in the models of agents and the environment. This type of modeling is considered as “bottom-up” since the emergence of global phenomena results from the individual agents’ behaviour and their interactions [10].

The proposed system in Chapter 3 consists of a decentralized online social network where there are thousands of entities (users, represented by their agents on clouds) exchanging information among each other, updating their internal states (relationship models) and as a result reducing the flow of irrelevant information, which is the emergent group-level phenomenon. Simulating such a system using ABM seems natural since each independent agent can represent a person in an OSN. Since the agents are acting autonomously without being controlled by a central component, the ABM approach fits with the proposed system’s decentralized architecture. Each agent implements rules determining which shared information from its friends is relevant by using the relationship model that resides in it. It also implements rules to ensure serendipity in the network by letting some popular but irrelevant information pass through. The social links between agents are maintained by the agents themselves by keeping records of all of their friends. Therefore, it is appropriate to simulate the proposed decentralized OSN using ABM.

The implementation of the simulation using ABM can be done in two ways:

1) Using special toolkits/libraries and development environment for ABM.
2) By implementing it from scratch using general programming languages, such as Python, Java, Erlang, and so on.

Some available toolkit/library-based development environments that have been used for ABM are Swarm, Repast, MASON, AnyLogic, NetLogo, and so on. According to the complexity and size of the simulation, the ABM simulation can vary from small (running on local computer) to large-scale distributed systems.

Swarm[4] [50] is one of the early platforms for implementing ABM. It is a collection of libraries which makes it easier for the developers to implement agent behaviour models and the environment for ABM. It was originally written in objective C and later a Java interface was added. Swarm has its own data structure and memory management to suit the development of ABM. It can be used to design ABM models for any domain. In contrast, Repast[5] [51] is an ABM development platform which is intended particularly for the social sciences domains (modeling populations of people). Repast has been implemented in Java and Python code, and can be integrated for model construction. Repast allows interactive mode of design, so it makes it easier for inexperienced users to develop behaviour models of agents. MASON[6] [52] is another ABM development environment, which is a hardware-invariant and faster alternative to Repast. MASON is capable of reproducing identical results from simulations run across different hardware platforms. MASON provides the functionality of stopping a simulation and continuing later on different computer which can be a useful option when running longer simulations. This is a domain-independent tool and can be used in any ABM problems. NetLogo[7] [53] is another available platform in which ABM can be implemented. NetLogo provides a powerful programmable modeling environment which is suitable to design complex systems. Its programming language consists of high-level structures and primitives that reduce the programming effort. NetLogo is designed especially for mobile agents that act concurrently on a grid space with behaviours defined by the local interactions between the agents over short

[4] http://www.swarm.org/
[5] http://repast.sourceforge.net/
[6] http://cs.gmu.edu/~eclab/projects/mason/
[7] http://ccl.northwestern.edu/netlogo/

interval of time. Another platform which supports ABM is AnyLogic[8] [54], which provides a visual language to simplify ABM development. AnyLogic uses UML (Unified Modeling Language) Statecharts for defining agent behaviours and Action Charts for defining algorithms.

All of the above development platforms are designed to make ABM development easier and faster. These platforms have implemented their own data structures and algorithms that are well-suited for ABM. But general programming languages such as Java, Python, Erlang, and others can also be used to implement ABMs which have special requirements, for example, a flexible topology of the agent network of connections, reasoning about connections, etc. These general programming languages should support threads which would hold the implementation details of the individual agent. The languages should also support data structure to hold the states of the agent and the environment. These programming languages should provide a way through which these threads can communicate since agents or threads should be able to communicate with each other. For the analysis of the run on the simulation, the data should be collected and should be stored in a database or file system. Therefore, these programming languages should provide a convenient interface with which data can be stored and viewed/accessed.

For the evaluation of the proposed approaches in Chapter 3, an ABM simulator modelling a real world decentralized OSN was implemented. The system implements relationship models and popularity parameters in social data for reducing the amount of irrelevant data received by users but ensuring the flow of serendipitous social data. The simulator has been developed in a general programming language – Erlang[9] [55]. Erlang is a parallel functional programming language which was designed for programming concurrent, real-time, distributed fault-tolerant systems [5]. The simulation of the proposed system consists of many agents representing users connected

[8] http://www.xjtek.com/
[9] http://www.erlang.org/

with each other in a social graph, acting independently. Each agent of the system is represented by Erlang process/thread capable of communication through its asynchronous message passing. The processes in Erlang do not share any memory and these processes are completely independent. This means a process running on one system can communicate with another process running on a different system. Erlang is designed to run in distributed multi-node environment which can be scaled up to incorporate large number of processes. In Erlang, a node is an Erlang abstract machine, which is capable of running a large number of processes. Erlang handles concurrency by running parallel activities on its own abstract machine rather than in the host operating system.

4.2 Simulation Design

The simulation of the proposed social network is implemented using the Erlang, a functional language. The main component of the simulation is an agent which represents a user in the social network. They also consist of a data container which has the agent’s relationship model regarding its friends, lists of friends, and its interest level in different areas of interest. The agent is responsible for sending out and receiving social data/messages to/from its friends. To be faithful to the decentralized architecture of the proposed OSN, the structure of the entire social network is not represented anywhere explicitly, but is encoded implicitly in the agents, through the agents’ friends lists and the relationship models, that is stored in the agents' databases. The combination of all the agents’ friends lists of all the agents represents the whole social network model that is simulated. A relationship model is maintained by the agent; it contains the relationship strength with respect to each area of interest with each of its friends. Figure 4.1 depicts a small community of users in an OSN which is represented by agents in the simulator. These agents are capable of sending and receiving social data between each other similar to real

OSN where users share social data in their friend circles. Each agent has set of rules (behaviour) which perform the filtering and sharing of social data to its friends. The flows of social data between friends/agents are bidirectional as shown by the bidirectional arrows in Figure 4.1.

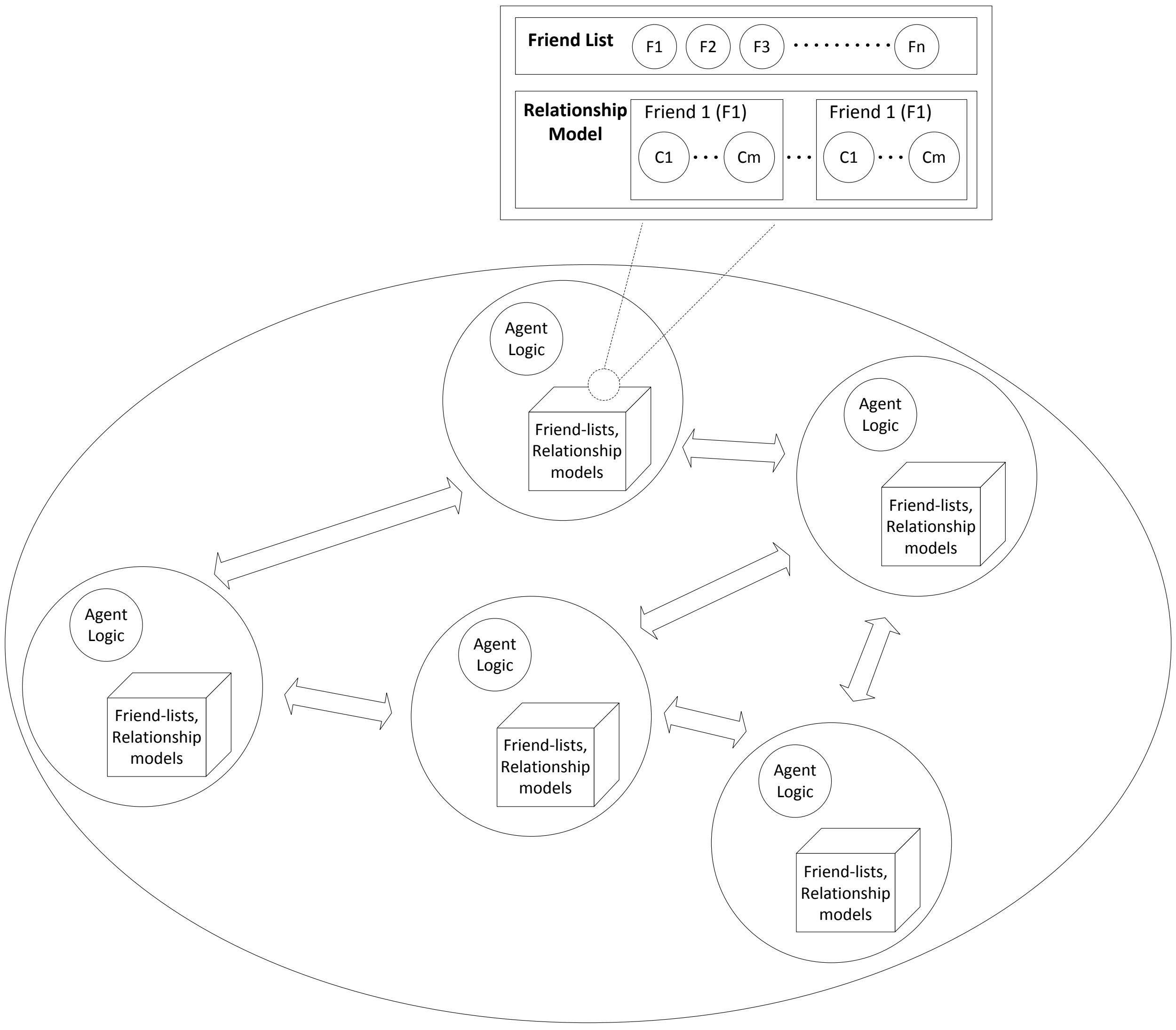


Figure 4.1: A community of agents in the simulator

4.2.1 An Agent's Behaviour

The heart of the simulation for the proposed system is the agents which represent the users of the real OSNs. Figure 4.2 depicts an agent's algorithm showing how it handles the incoming social data and shares it to its connected friends.

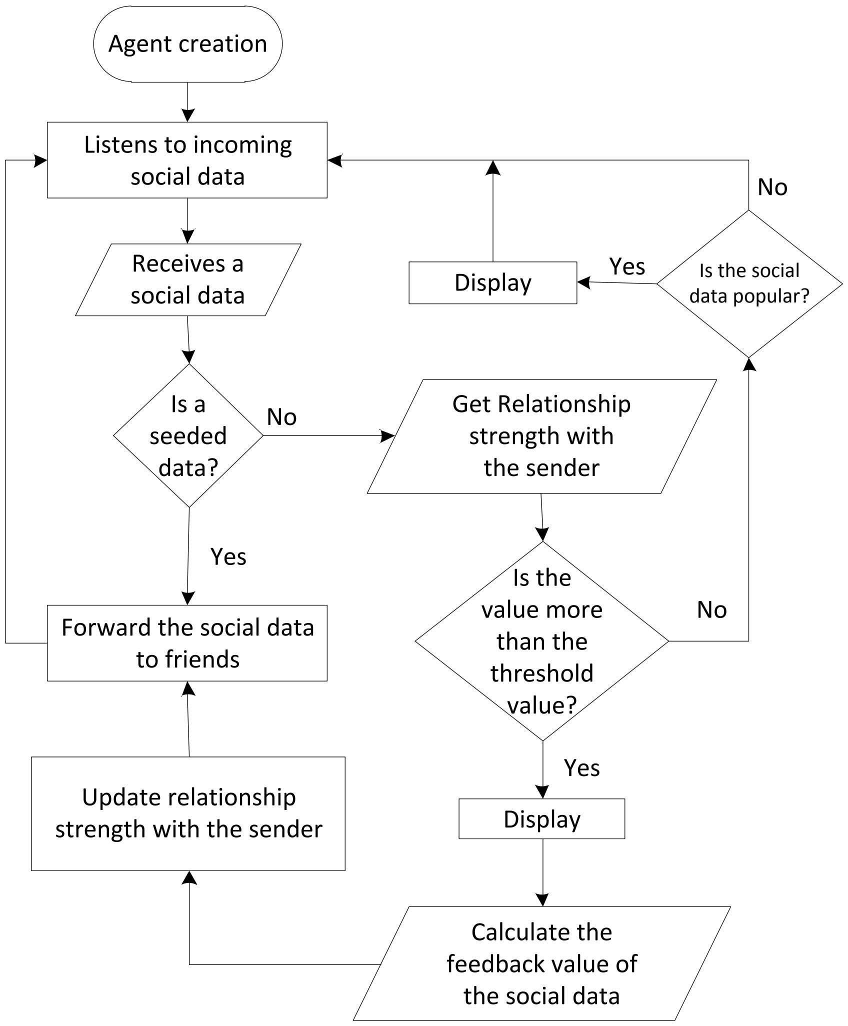

Agent creation
Listens to incoming social data
Receives a social data
Is a seeded data?
No
Yes
Get Relationship strength with the sender
Is the value more than the threshold value?
No
Yes
Display
Is the social data popular?
Yes
No
Display
Forward the social data to friends
Update relationship strength with the sender
Calculate the feedback value of the social data

Figure 4.2: Algorithm/behaviour of an agent

After the creation of an agent, its sole purpose is to listen to the incoming social data and act accordingly. Incoming social data can be of two different types: 1) seeded from the system (simulating users discovering new items of interest that they share with their friends) and 2) forwarded by friends (simulating users "re-sharing" items from friends that they found interesting). Seeded social data are injected by the system into the network so as to simulate the discovering of new information by the users of the OSN. Seeded social data are generated in a regular time interval and belongs to randomly chosen areas of interest. The social data are seeded in regular interval of time to a subset of agents of the network. In this way, the process of users posting various new social data to their friends is simulated. If the incoming data is of type 1) i.e. new seeded data, the agent always forwards it to its friends. If the incoming data is of type 2) i.e. forwarded by friends, it would analyse the data and act accordingly. According to the agent's algorithm (process), first it will evaluate whether or not the relationship strength with the sender of the data in the area of interest in which the social data is classified is high enough for the agent to view that social data. If its value is low then the social data should be ignored i.e. filtered away, unless it is popular data (which could be serendipitous for the user). So the agent will check the popularity value of the social data and if it is high then it is displayed to the user. If the popularity value is low, the social data is filtered out by the filtering mechanism. It is noteworthy, that the relationship model is not updated in both cases, i.e. whether the data was filtered away or shown to the user as potentially serendipitous data; it would not impact the strength of the relationship with the sender of the data.

If the relationship strength value is high enough, the social data will be displayed to the user (in the simulation, of course, it is not displayed to the agent), and feedback for that social data is generated according to the interest level of the agent in the semantic area of the social update. Then the relationship strength between the agent and its friend is updated using equation (3.1) and stored in the agent's memory. If the feedback value is 0.9 according to table 3.1, then the agent forwards this social data to its friends. The simulation maintains the order of arrival of social data, processes the feedbacks and updates of the relationship model in first come first serve manner.

In summary, the simulation represents the OSN implicitly through the many agents representing users in the OSN. Each agent stores data which enables the functioning of the OSN:

- each agent has its own list of interest values according to a predefined set of different areas of interest, common for all agents in the system,
- each agent has its own list of friends,
- each relationship with a friend is represented by a set of values that express the strength of the relationship with respect to the different areas of interests represented in the system.

Each agent maintains all these values in its memory at runtime, and in the end of the simulation they are written into a file.

4.2.2 Social Network Dataset Used in the Simulation

The social network or graph which is required to initialize interactions between agents in the simulator is based on a real social network, and generated using subset from the StudiVZ[10] [20] dataset. This dataset is a publicly available dataset from a popular German social network for students with over 1 million user profiles. The dataset contains anonymous user id along with friends' ids in tab separated format. A program is written in Python[11] [56] to parse through the file containing the StudiVZ dataset. This program goes through the given raw data from StudiVZ and creates various subsets of it. For the experiments 6 subsets were selected that contain 427, 1300, 2613, 5324, 7062, and 9094 unique user profiles. For the creation of a subset, the following algorithm was used:

1) Select the top 5 or 10 user profiles (nodes) with highest number of friends, to ensure that the subsets will have a larger number of connections.

[10] http://studivz.irgendwo.org/
[11] http://www.python.org/

2) Do breadth-first search on each of the friends of these selected nodes until a certain number of friends is reached (usually a smaller number leads to smaller community).

4.3 Implementation Details

From the initial preparation of the social network that will be used in the model to the analysis of the results after the run of the simulation, the process consists of four stages:

1. Parsing and preparation of the subset graph
2. Distribution of interests over the nodes
3. Simulation
4. Analysis phase

Each of the stages is implemented using different techniques.

4.3.1 Parsing and Preparation of the Subset Graph

Data from the StudiVZ are available in tab-separated values (tsv) format starting with the node's unique id and following in the same line, separated by tab character, are node's friends' unique ids. For the analysis of the generated sub-graph a python library called NetworkX[12] [57] is used. This library is used to identify the number of nodes, edges and the clustering coefficient of the sub-graph. The clustering coefficient determines how much nodes are clustered together in a graph. Code snippets for parsing the dataset, creating subset of social graph, writing this subset into file, and calculating parameters of the social graph are given in Appendix A.

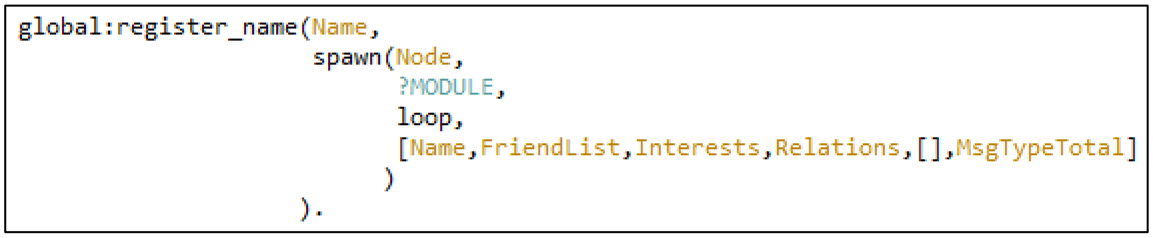
[12] http://networkx.lanl.gov/

4.3.2 Distribution of Interest

Using the sub-graph from the above step, the interest values (which determines how much a node is interested in a category) of each node in a given number of categories (constant) is distributed in the range of [0, 1] using power law. The power law distribution means that 80% of the nodes will be interested in a popular category, and the popular categories are only 20% of the categories. The interest distribution over the nodes is written in a tab-separated values (tsv) format file with initial value - the node id and followed by the list of values of interest in each of the given categories serially. This step is implemented in Python programming language and code snippet is provided in Appendix A.

4.3.3 Simulation

Each agent is created as a process in Erlang. A code snippet of an agent creation in Erlang is shown below in Figure. 4.3.

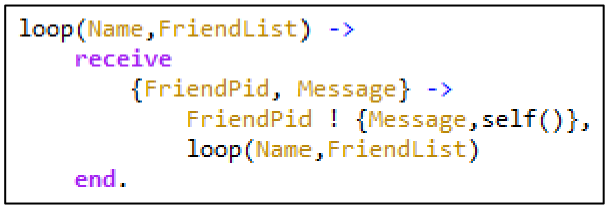


```
global:register_name(Name,
                     spawn(Node,
                           ?MODULE,
                           loop,
                           [Name,FriendList,Interests,Relations,[],MsgTypeTotal]
                          )
                    ).
```

Figure 4.3: Process creation in Erlang

Processes in Erlang are created by evaluating *spawn* as shown in the above snippet. This *spawn* function has four parameters. The first parameter is the name of the Erlang node on which the process should be created. The second is the name of the module on which the implementation of process should reside. The third is the name of the function in which implementation of process should reside. And the fourth parameter is the parameters to the agent function. The parameter to the agent function defines the properties of that agent – its name, its friends, interest values on areas of interest, relationship models for friends, and the kind of message it is getting. These data are stored in the agent itself and not stored in file system or database in runtime so that the execution of the simulation does not depends on the read/write functionality. After the creation of a process, *spawn* returns back the process identifier (*PID*) of the process created. The process is then registered in global namespace so that it can be accessed using the process name instead of *PID* by evaluating *global:register_name()*.

Each agent in the simulation shares data through a message passing mechanism provided by the Erlang language. The messages are passed between processes/agents by evaluating send (implementing using infix operator – !) to send a message and *receive* to receive a message as shown in the following code snippet (Figure 4.4). The implementation of the agent for the simulation is given in Appendix B.

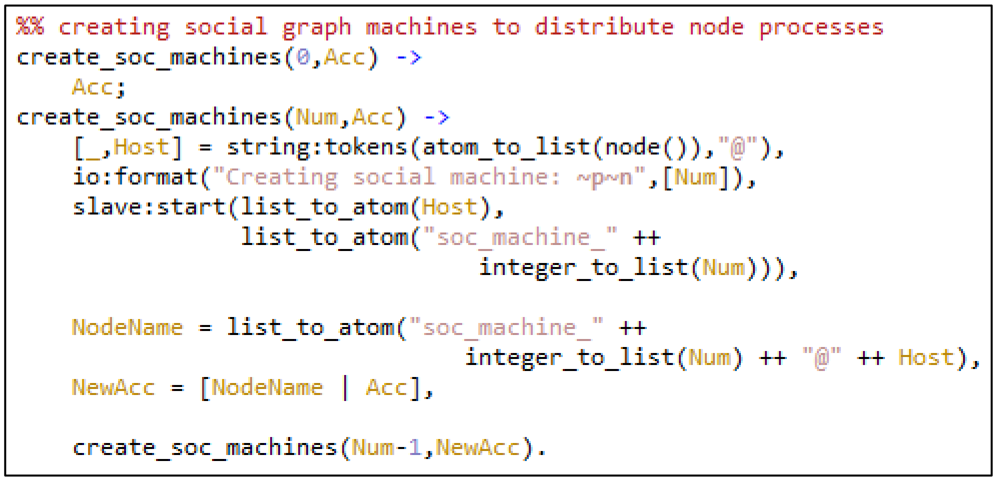


```
loop(Name,FriendList) ->
    receive
        {FriendPid, Message} ->
            FriendPid ! {Message,self()},
            loop(Name,FriendList)
    end.
```

Figure 4.4: Process implementation in Erlang

Each social data in the simulation is implemented as a process and the information related to it is stored in the process itself. This helps in reducing the read/write functionality to the database or file system, hence making the simulation run faster. At the end of each run, data from these processes (agents and messages) are collected and are written in comma-separated value (csv) format. The implementation of the message process used to store information regarding the messages flowing in the network and the implementation responsible for writing data into csv format at the end of simulation are shown in Appendix B.

The agents in the simulator can be distributed in different Erlang abstract machines and the number of machines can be specified during the simulation's configuration. For the configuration of simulation, a web interface is also designed and it can be done by invoking a function directly. Figure 4.5 describes the code snippet that shows how multiple Erlang abstract machines are created dynamically as a slave node. Slave nodes are invoked by evaluating *slave:start()* as shown below.



```
%% creating social graph machines to distribute node processes
create_soc_machines(0,Acc) ->
    Acc;
create_soc_machines(Num,Acc) ->
    [_,Host] = string:tokens(atom_to_list(node()),"@"),
    io:format("Creating social machine: ~p~n",[Num]),
    slave:start(list_to_atom(Host),
                list_to_atom("soc_machine_" ++
                                 integer_to_list(Num))),

    NodeName = list_to_atom("soc_machine_" ++
                                integer_to_list(Num) ++ "@" ++ Host),
    NewAcc = [NodeName | Acc],

    create_soc_machines(Num-1,NewAcc).
```

Figure 4.5: Erlang slave machine creation

4.3.4 Analysis Phase

After the complete run of the simulation, the data are collected in csv format. Since the raw data in csv format contains each interaction of the agents with each other, the data is processed using Python programming language to transform into simpler form to create tables and graph. Graphs are plotted using GNUPLOT[13] [58].

[13] http://www.gnuplot.info/

4.4 Summary

In this chapter, we discussed the design and implementation of a simulation which is used to evaluate our approach and test the hypothesis that it helps to reduce the propagation of irrelevant information while keeping a flow of serendipitous information in the decentralized social network. The next chapter presents the experiments conducted using this simulation, the results and analysis of the results using tabular and graphical representations.

CHAPTER 5
EXPERIMENTS AND EVALUATIONS

Experiments were conducted using the simulation presented in chapter 4 and the results are presented in this chapter. Section 5.1 describes the social graph used in the experiments. Section 5.2 focuses on the parameters that were considered when executing the simulation. Section 5.3 describes goals of the experiments conducted. Section 5.4 describes experiment setups for the simulation to run with specific parameters and conditions. Section 5.5 presents the results from the experiments and Section 5.6 contains the discussion of the results.

5.1 Social Graph

As explained in Chapter 4 the social graph that is used to simulate the decentralized OSN is a subset from the StudiVZ dataset. An enterprise or neighbourhood social network (the most likely adopters of decentralized OSN) is most likely to be smaller. That is why several subset graphs are generated from the dataset for the experiments. To determine the connectivity between nodes in each of these sub-graphs, the average clustering coefficient was calculated. The clustering coefficient determines how much nodes are clustered together in a graph. A *Clustering coefficient* of a graph is defined as the number of edges between the nodes over the total number of possible edges. The sub-graphs that are used in the evaluation have an average clustering coefficient ranging from 0.13 to 0.17. This clustering coefficient is in the range of the typical coefficients for OSNs. For example, Facebook's clustering coefficient varies between 0.133 and 0.211, and the average over all 22 largest regional networks in Facebook is 0.167 [47]. The analysis of Facebook data shows that nodes with lower degree have higher clustering coefficients [47]. That means that the users who have fewer friends on average will be part of denser clusters, as their friends are more likely to be interconnected. The overall picture of a network with 3324 nodes, 4383 edges and clustering coefficient of 0.13, which is one of the dataset on which experiments were done and discussed later, is illustrated in Figure 5.1 which is drawn using Pajek[14] [59].

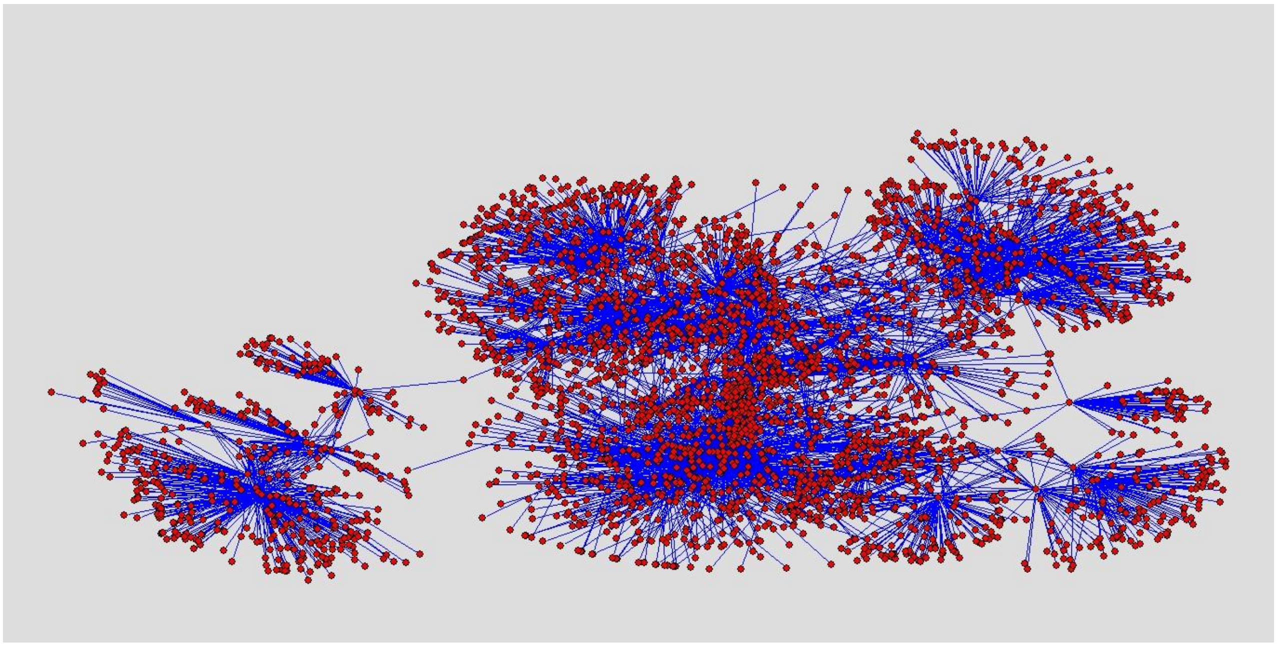


Figure 5.1: Subset from StudiVZ dataset (visualized using Pajek [59])

5.2 Parameters

Ten different interest categories are distributed over the nodes following a power law. Here the agents are referred to as nodes in the social graph which are connected to each other through social relationship (the edges in the graph). For the simulation of social updates being generated or shared by users, 10,000 messages are seeded to the network. To make the message distribution

[14] http://vlado.fmf.uni-lj.si/pub/networks/pajek/

realistic, 25% of the population are selected for each seeded messages for seeding into the network, since, according to [15], approximately 28-29% of the population of Twitter updates their status regularly. The seeded nodes are selected in such a way that first priority is given to those nodes whose interest matches the category of the message and only when there is not a sufficient number of interested nodes - to other random nodes. The only randomness in the simulation comes from this selection of random nodes otherwise all other parameters are set by the experimenter. Each node forwards each seeded social update to its friends. A node that receives a social update does the following:

1. It checks the relationship value with the sender on the category of the message. If this value is too low, it does not do anything (in the real system it would not display the message to the user) and does not forward it further.
2. If the relationship value is high enough, it checks its own interest level in this category and updates its relationship strength with the sender in this category. Different values of feedback for updating the model are used depending on the level of interest that the node has in the category. If the level of interest is high, then the relationship will be strengthened significantly, and the social update is forwarded to its friends. If the node's level of interest in the category is in the medium range, the relationship will be slightly strengthened (simulating that the user has provided a rating or a comment). If the level of interest is low (i.e. the user has just viewed the update), the relationship strength slightly decreases. After a certain number of such received marginally interesting messages, the strength of the relationship in this category drops down below the threshold and future messages from this node are filtered away.

3. To ensure flow of serendipitous updates through the network, popularity is taken as parameter and depending on the value of this parameter, the message can by-pass the filtering mechanism. Even if the strength of relationship in a category is low, due to the popularity of the message/update, it will not be filtered away. As mentioned in Section 3.2.3, popularity is considered in two ways: global and local popularity. Both are evaluated with experiments in the following sections.

Initially, the relationship strengths for all nodes and categories are set to the maximum value of 1. This ensures the complete flow of information at the beginning of the experiment. After that, the system will gradually learn as the relationship model is updated by the recipients of the social updates. The learning rate of the relationship model (i.e. the reinforcement learning formula) is set high enough so that the network can learn fast. However, the relationships should not deteriorate too quickly, to preserve the flow of information. The rationale for choosing the learning rate value is presented in chapter 3.

5.3 Experiment Goals

To evaluate our proposed approach, we need to test the following hypotheses:

1. With the seeding of more and more social data, the network will learn to filter out irrelevant information, so the nodes will receive less irrelevant social data.
2. Relevant information is reaching out to all the nodes interested in the category of the information.
3. Serendipitous information continues to flow in the network.

5.4 Experiment Setups

Experiments were conducted in three different local machines according to the size of social graph. Smaller social graph required smaller amount of RAM, while bigger graphs required machines with larger RAM.

A specification of the environments used in the experiments is given in Table 5.1:

Table 5.1: Machines specification for experiments

Operating System	**Processor**	**RAM**	**Erlang Shell Version**
Windows 7 Enterprise 32-bit version	Intel® Core™ i5 CPU 650 @ 3.20 GHz	4 GB (2.99 GB usable)	5.9
Windows 7 Enterprise 64-bit version	Intel® Core™ i7-2620M CPU @ 2.70 GHz	8 GB	5.9
Windows 7 Enterprise 64-bit version	Intel® Xeon® CPU 5140 @ 2.33 GHz	16 GB	5.9

Six different sub-graphs of social graph from StudiVZ were selected for the experiments. Table 5.2 have listed the properties of the selected subsets of social graph.

Table 5.2: Properties of selected subsets of social graph

Number of nodes	**Number of edges**	**Clustering coefficient**
427	491	0.136
1300	1671	0.174
2613	3214	0.145
5324	7364	0.130
7062	10536	0.126
9094	13631	0.125

Interest distribution based on power-law distribution is also calculated for 10 different areas of interest for the above sub-graphs. Each experiment is repeated for 10 times to verify that results are reproducible. The average of all the 10 repeated experiments and the standard deviation is calculated and presented. The following sections presents the experiments conducted.

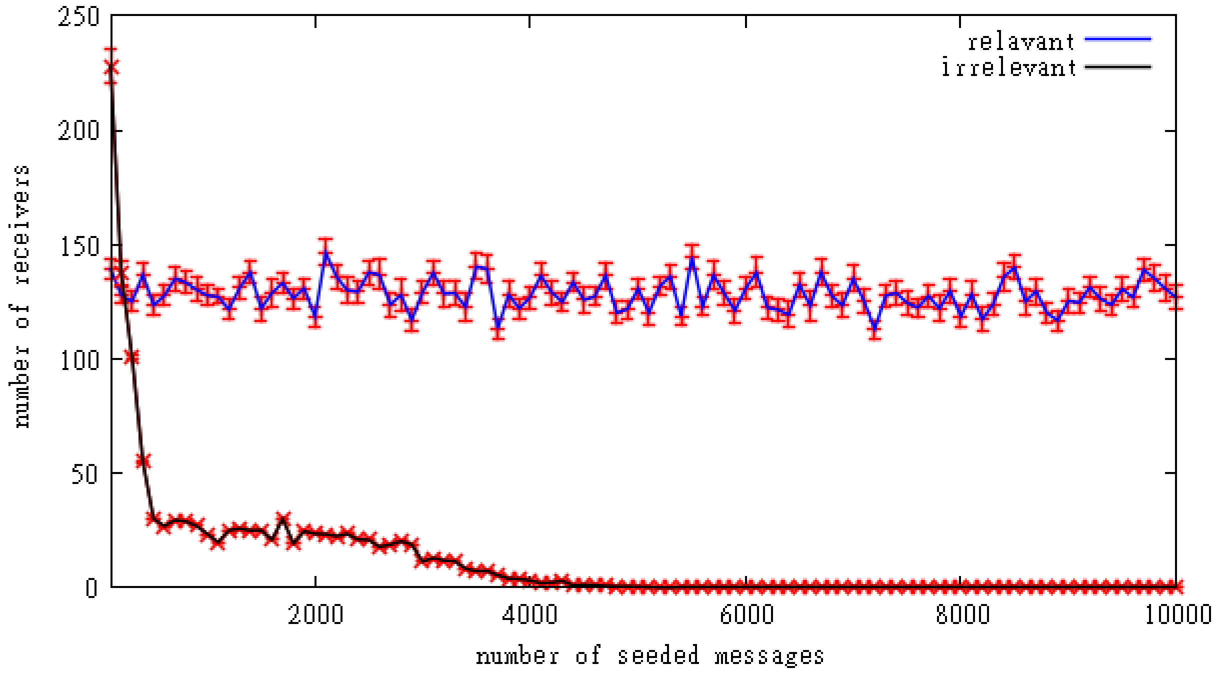
5.5 Experiments

This section discusses experiments that were conducted on the social sub-graphs that were listed in Table 5.2.

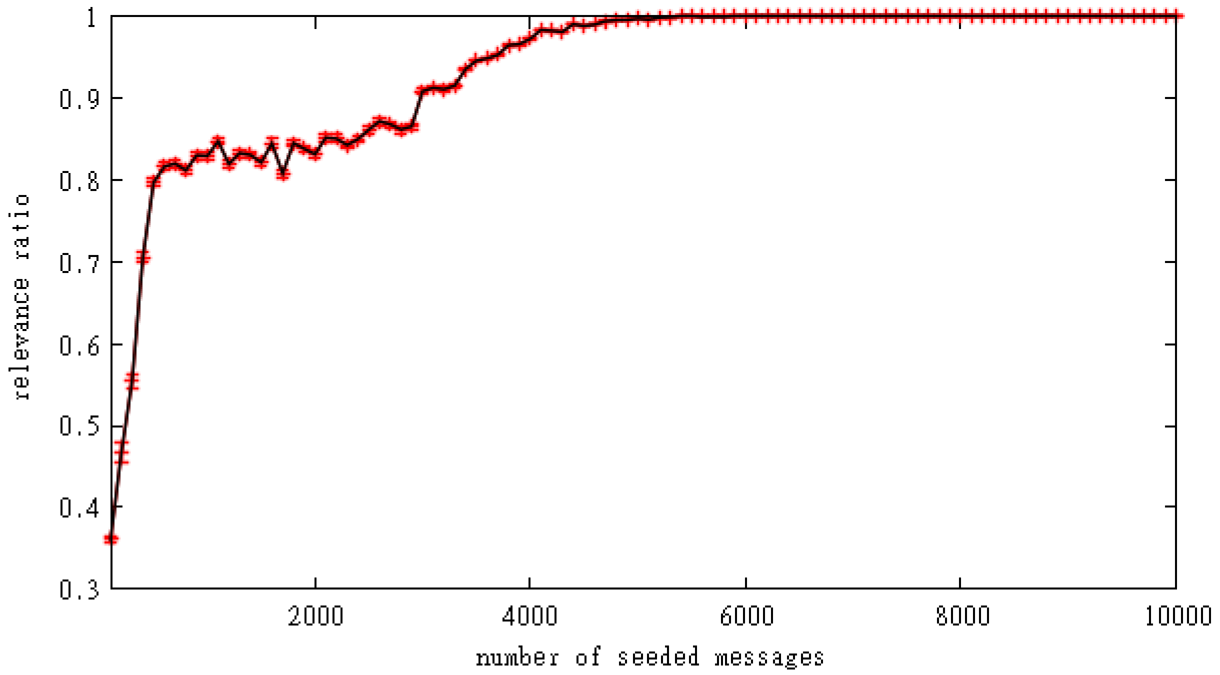
5.5.1 Experiment 1: Does Relationship Model Filter Out Irrelevant Social Updates?

The following experiment was conducted to verify that the relationship model filters out irrelevant social updates from the network. This experiment does not include serendipitous information flow using popularity parameters. The results from the experiments on subsets of dataset listed in Table 5.2 are illustrated below from Figure 5.2 to Figure 5.7 in ascending order according to the size of the subsets. Each data point on the graph is the mean of 10 runs of the experiment for the same social graph. The average value and the standard deviation over the 10 runs are plotted, where black and blue colors represent average value, while red color represents standard deviation.

Each illustration has two graphs: a) shows the number of nodes receiving relevant messages and the number of nodes receiving irrelevant messages at given number of seeded messages, b) shows the relevance ratio. The relevance ratio is the number of nodes to which the message forwarded is relevant out of total number of nodes that received the same message.

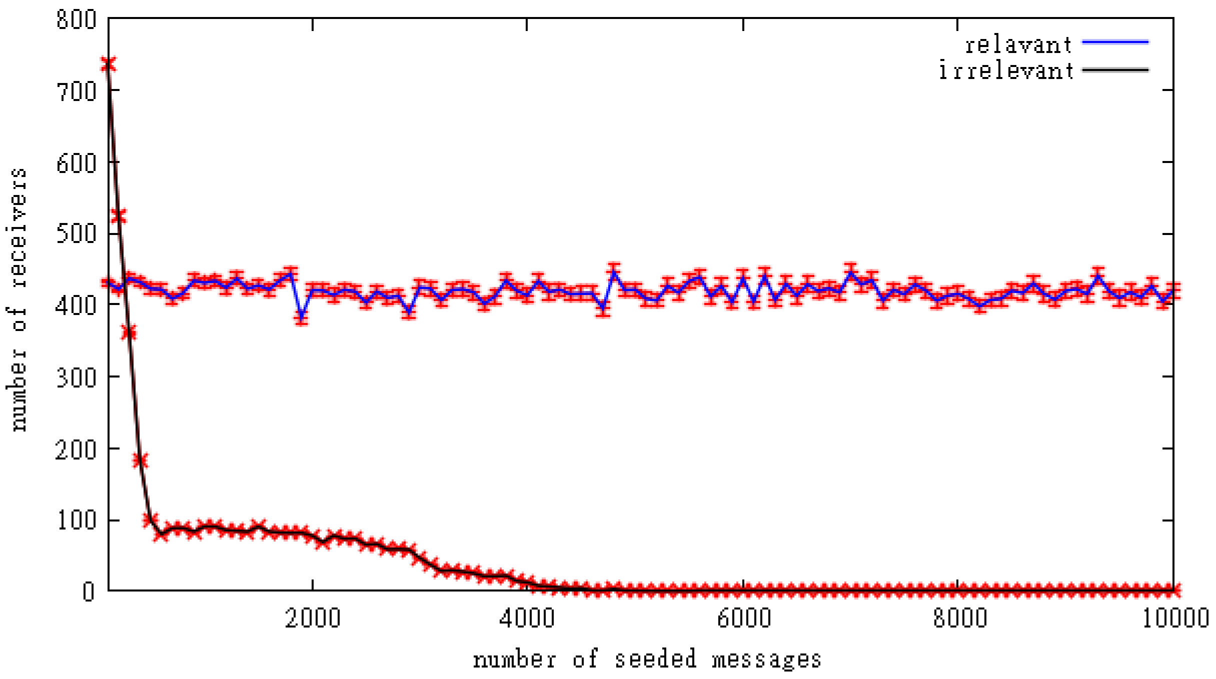


a) Flow of relevant vs. irrelevant messages through the network for each seeded messages

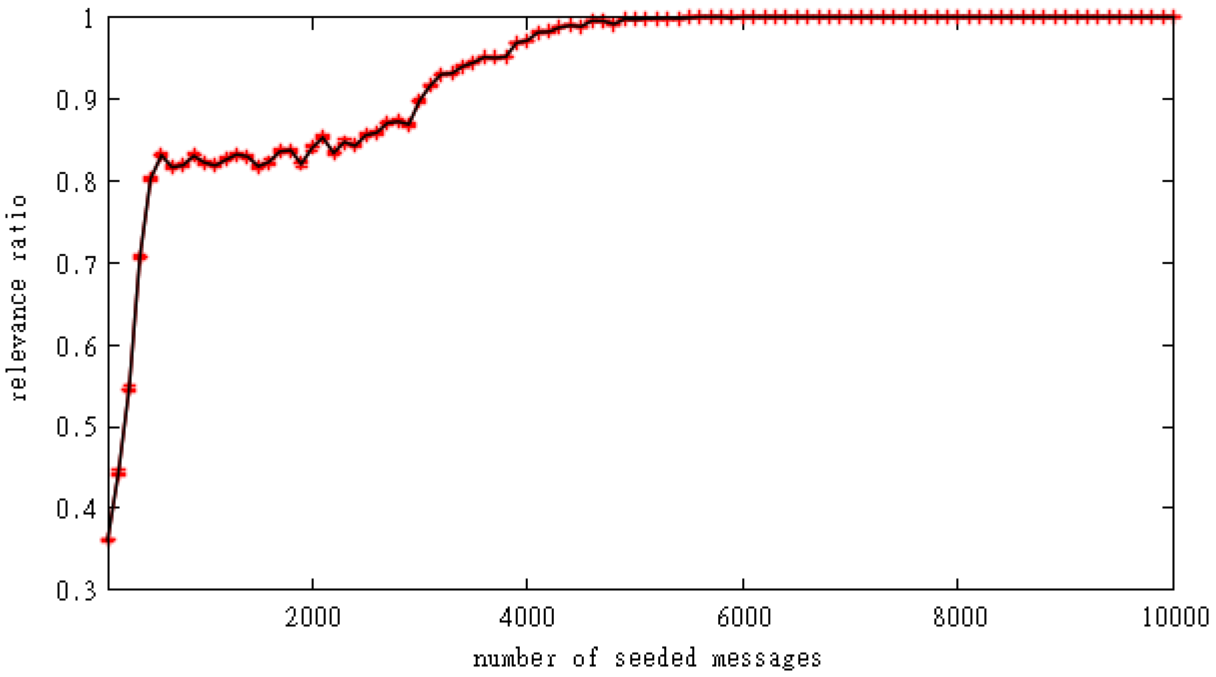


b) Relevance ratios of seeded messages

Figure 5.2: Result of Experiment 1 for subsets of 427 nodes

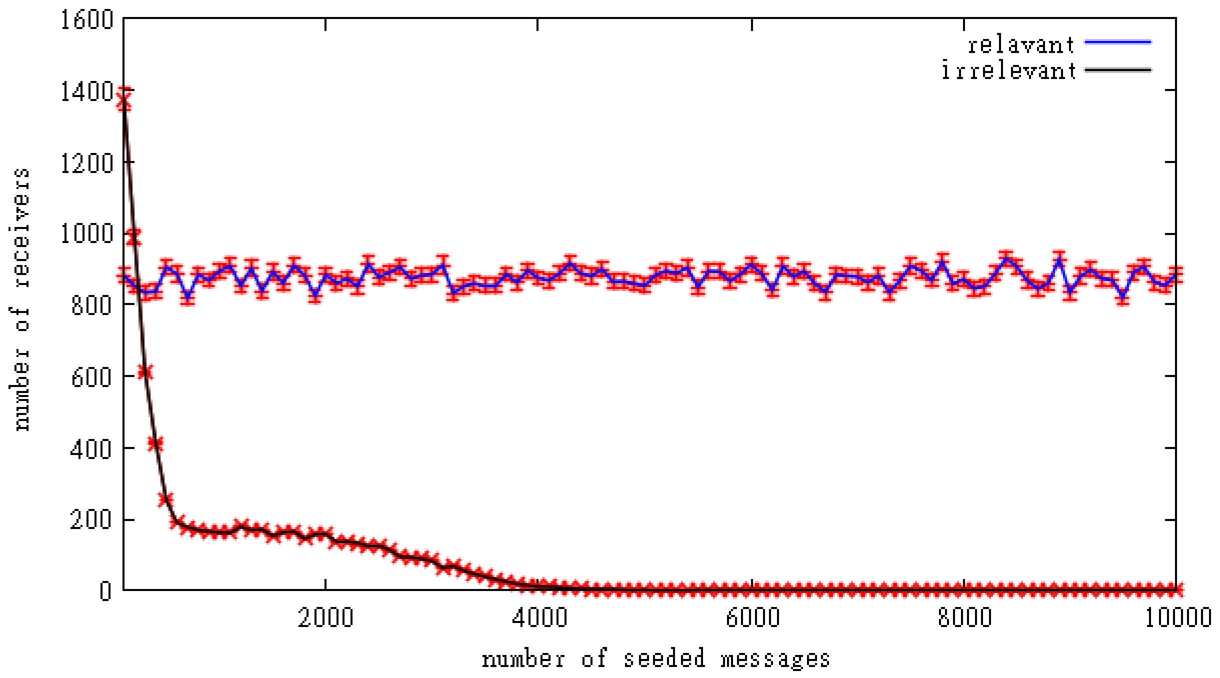


a) Flow of relevant vs. irrelevant messages through the network for each seeded messages

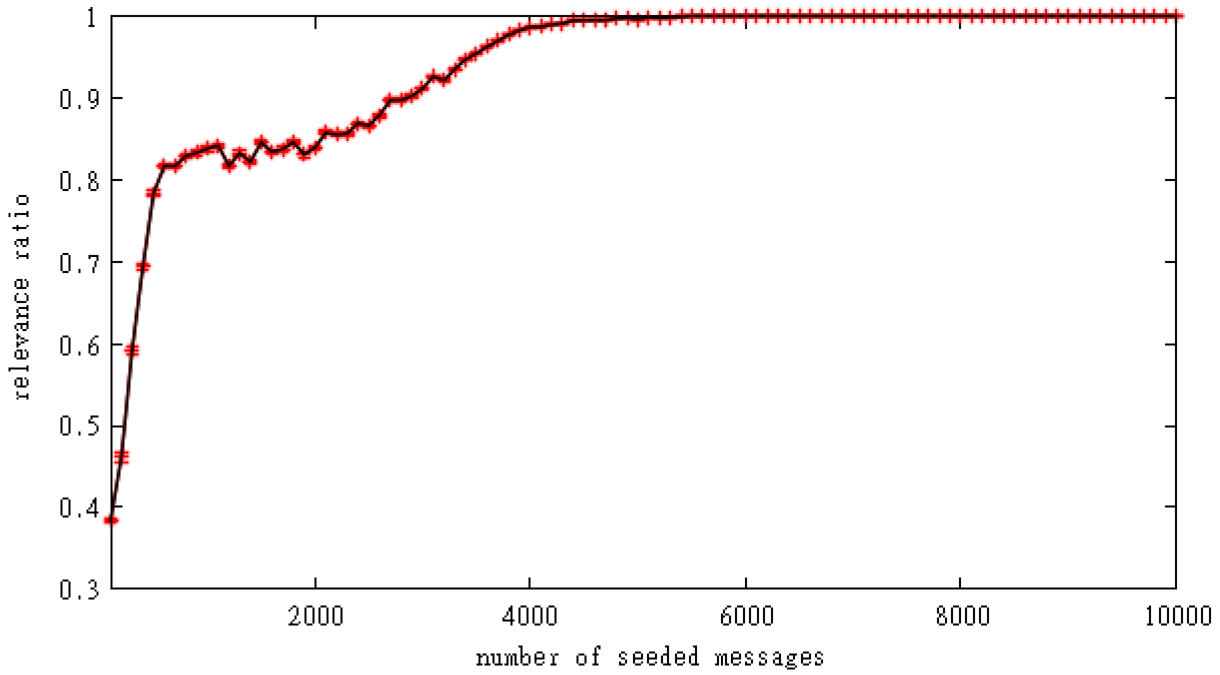


b) Relevance ratios of seeded messages

Figure 5.3: Result of Experiment 1 for subsets of 1300 nodes

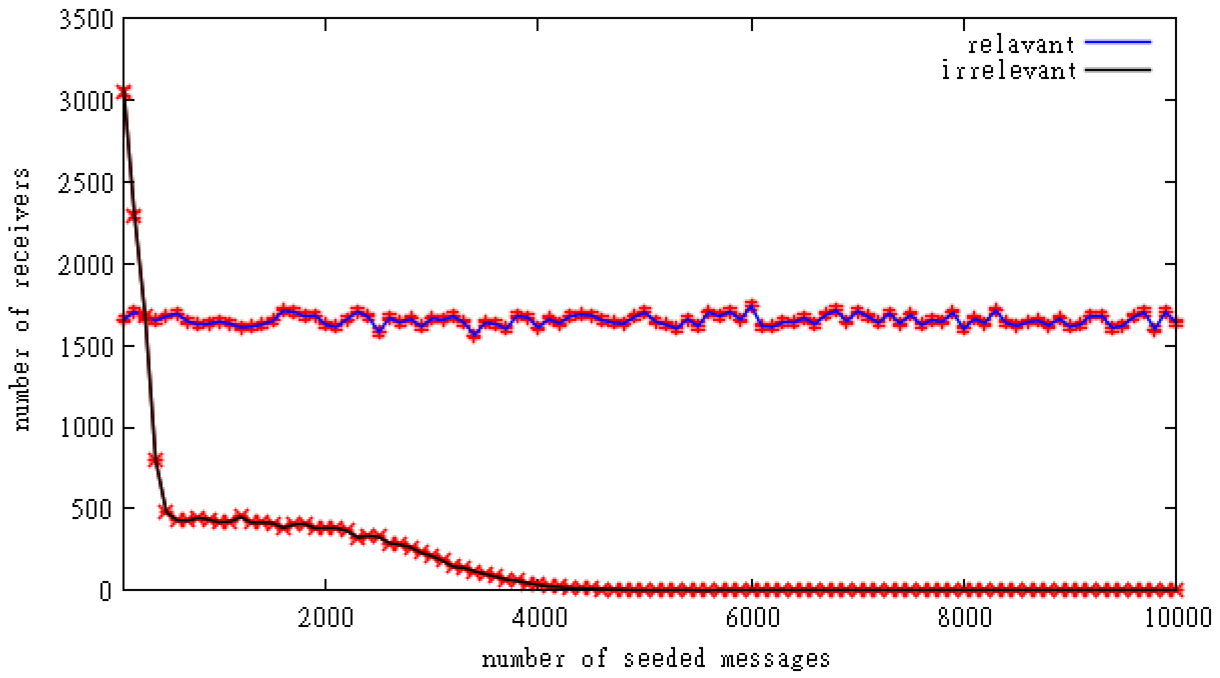


a) Flow of relevant vs. irrelevant messages through the network for each seeded messages

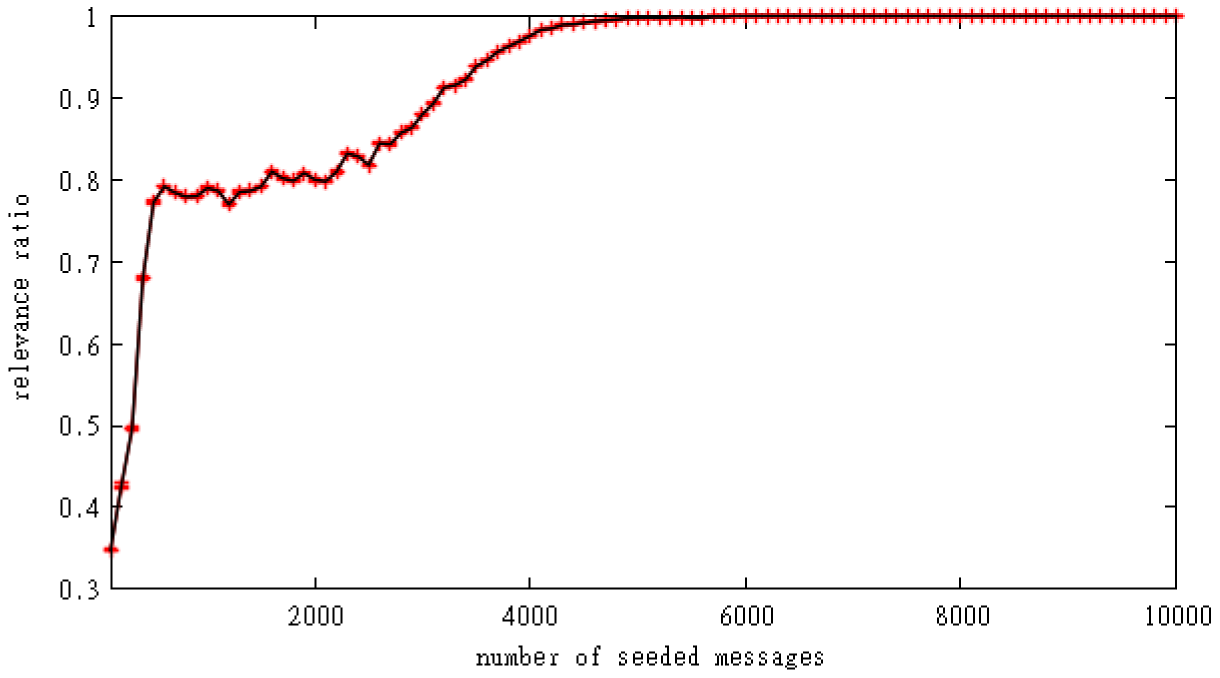


b) Relevance ratios of seeded messages

Figure 5.4: Result of Experiment 1 for subsets of 2613 nodes

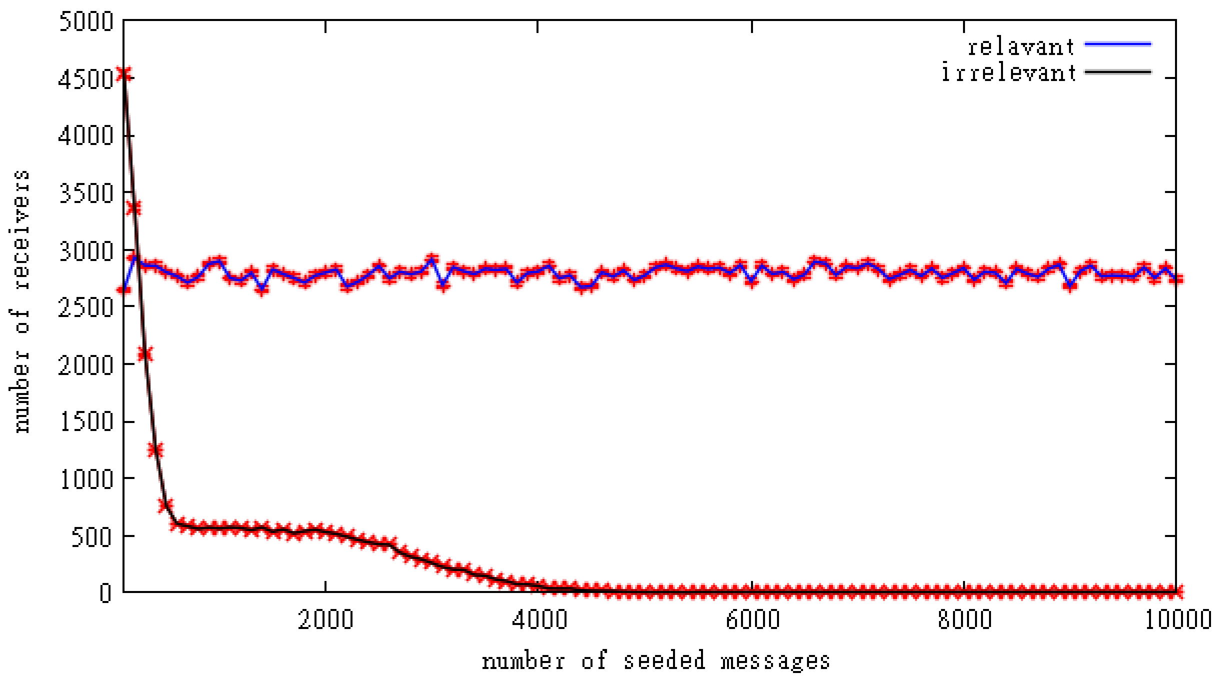


a) Flow of relevant vs. irrelevant messages through the network for each seeded messages

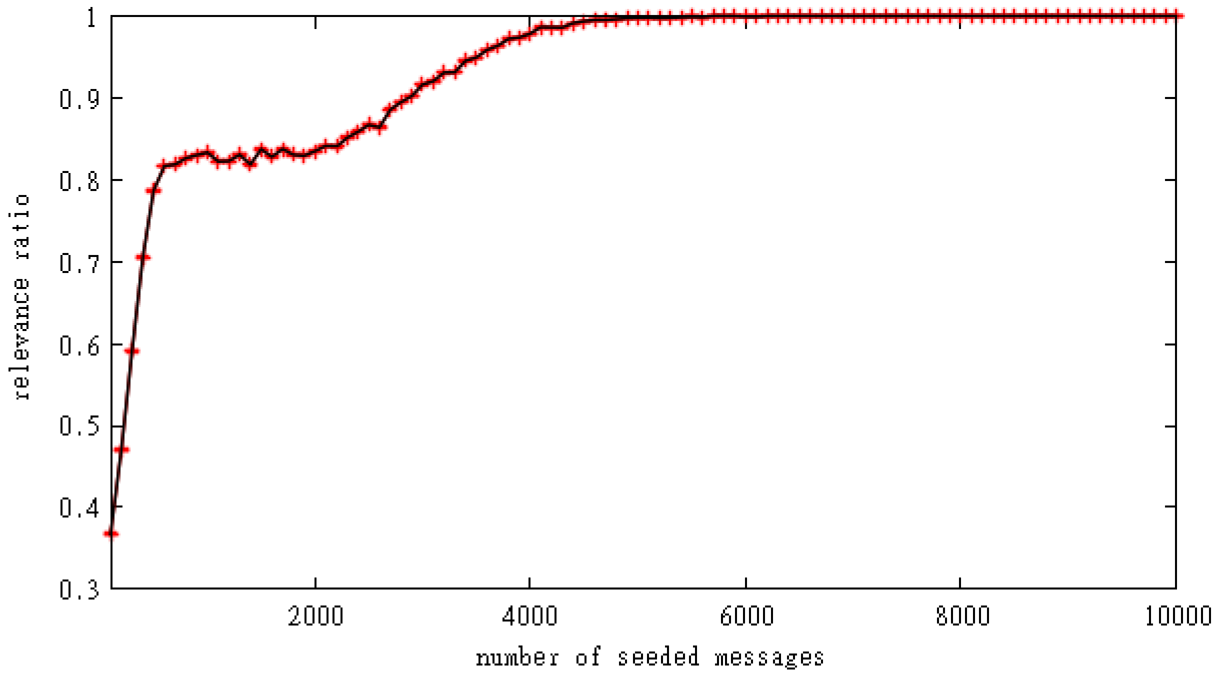


b) Relevance ratios of seeded messages

Figure 5.5: Result of Experiment 1 for subsets of 5324 nodes

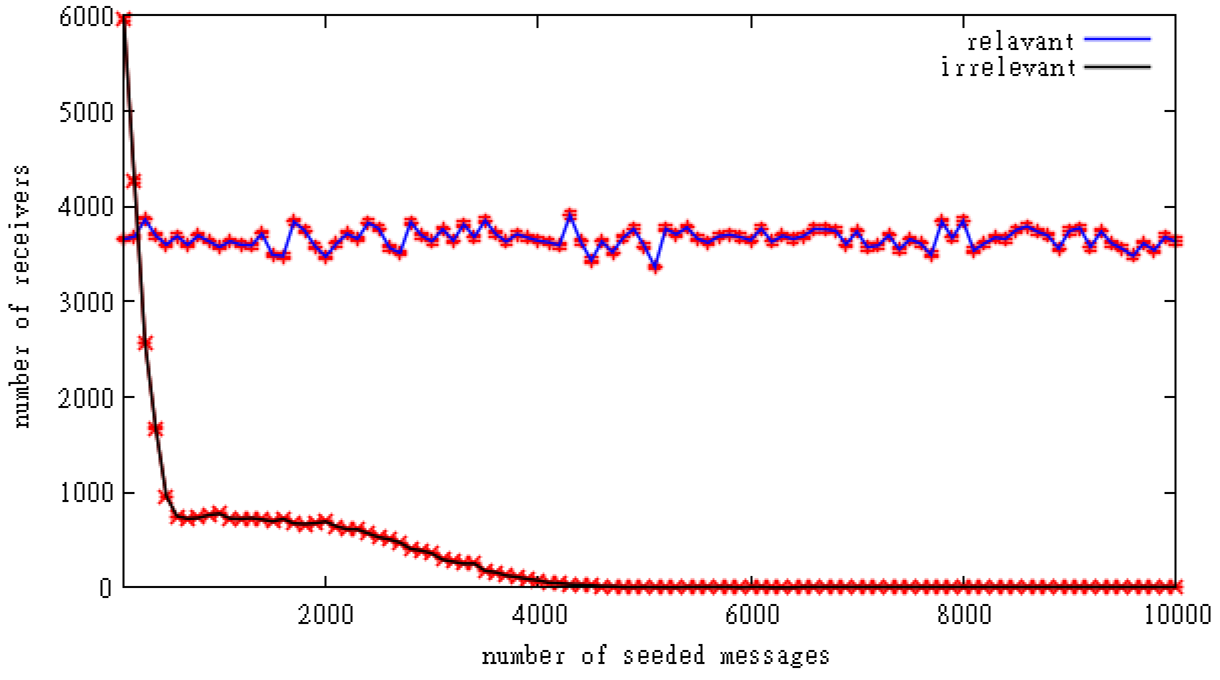


a) Flow of relevant vs. irrelevant messages through the network for each seeded messages

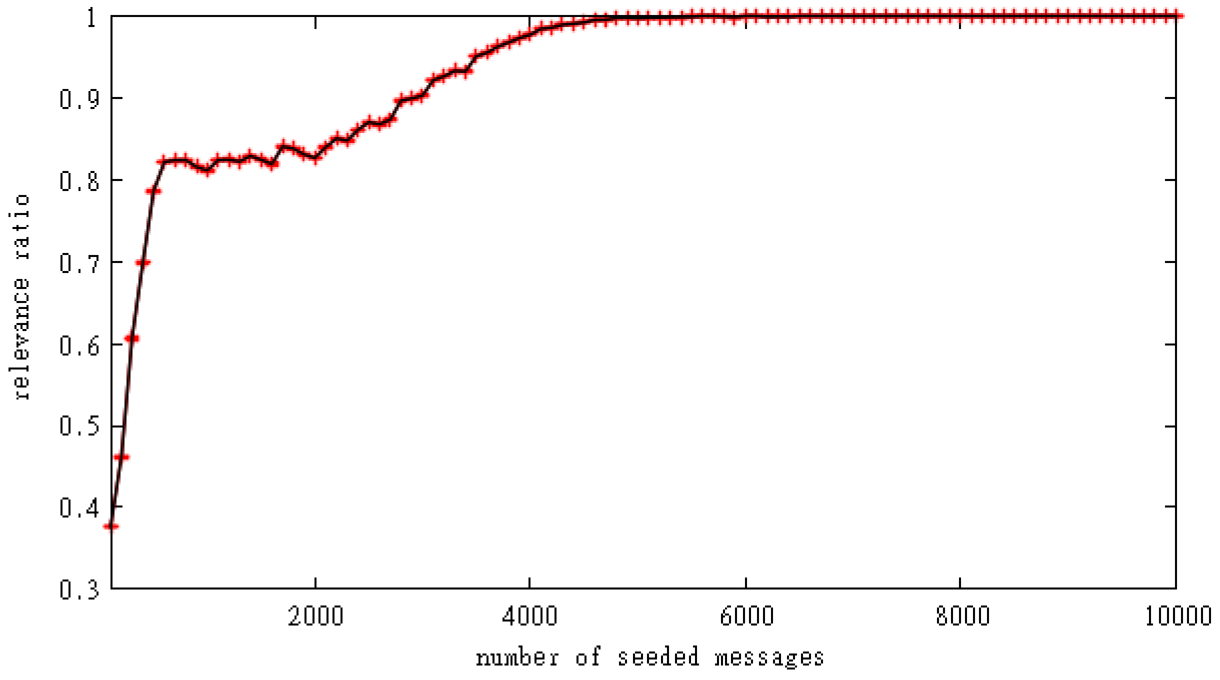


b) Relevance ratios of seeded messages

Figure 5.6: Result of Experiment 1 for subsets of 7062 nodes



a) Flow of relevant vs. irrelevant messages through the network for each seeded messages



b) Relevance ratios of seeded messages

Figure 5.7: Result of Experiment 1 for subsets of 9094 nodes

From the illustration a) of results for different social network subsets in Figure 5.2 to Figure 5.7, a convergence of the irrelevant messages/updates down to zero is observed at around 4000 seeded messages for all of the experiments. The size of the subset has almost no effect on the rate of reduction of irrelevant messages. The standard deviation is lower for the irrelevant messages than for the relevant messages. This is because the mechanism is filtering out irrelevant messages out of the network while preserving the relevant ones.

From the illustration b) which shows the change of the overall relevance ratio of received messages by the nodes for no. of seeded message (with the flow of seeded messages), we can observer a steep increase at the beginning of the experiment; this is due to the fact that at the beginning of the experiment all the nodes in the social network have the highest relationship strength with their friend nodes. Therefore, at the beginning of the experiment all messages are shared, also with nodes that have low interest level in the message. But as the system learns by each node updating its relationship model, later arriving irrelevant information is filtered out and thus reaches lesser population in the social network, interested in the category of the information. This confirms the first hypothesis, that with the passage of time, users will only see relevant social data in their social network.

The following Table 5.3 shows the total flow of irrelevant and relevant messages through the simulation during its run and how many of irrelevant messages are filtered. It can be seen from the table that most of the irrelevant messages are filtered out by the mechanism. This validates the hypothesis that on reduction using the relationship model as filtering mechanism reduces the number of irrelevant messages.

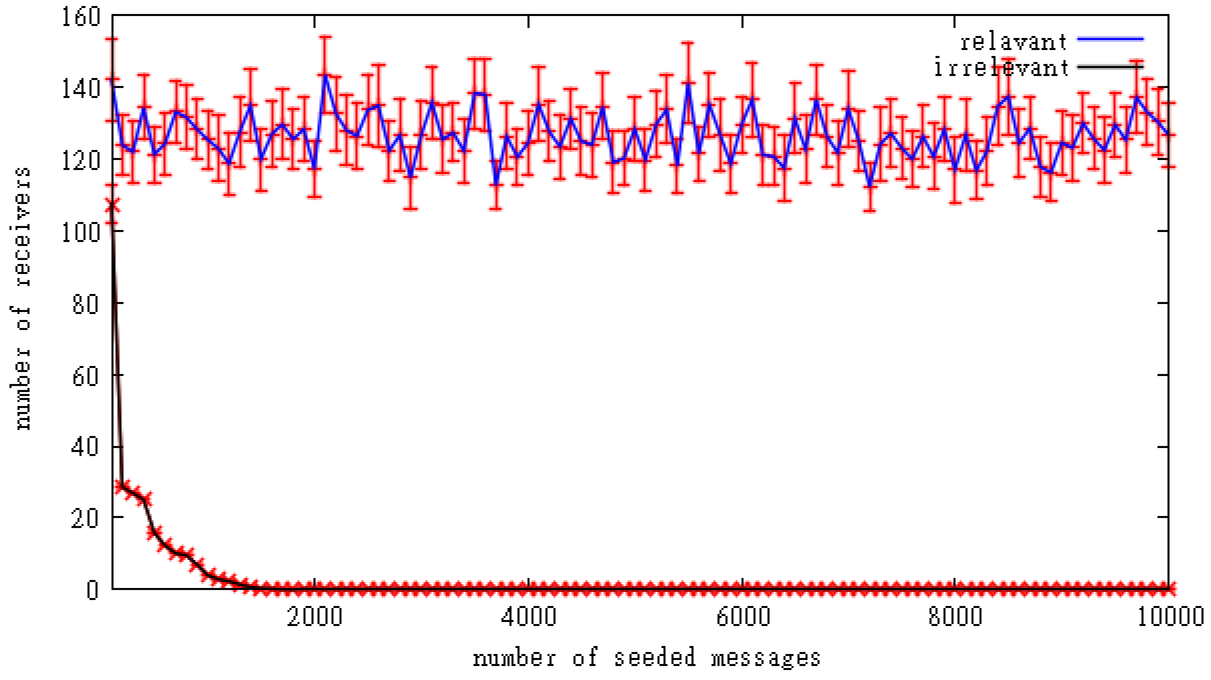
Table 5.3: Total flow of irrelevant, relevant, and filtered messages

Nodes	Irrelevant	Relevant	Filtered
427	2,238,797	1,283,691	2,117,596
1300	7,284,753	4,192,060	6,875,892
2613	13,939,482	8,754,218	13,171,407
5324	30,770,434	16,555,656	28,902,506
7062	48,253,665	27,931,151	45,669,114
9094	61,648,559	36,614,571	58,316,719

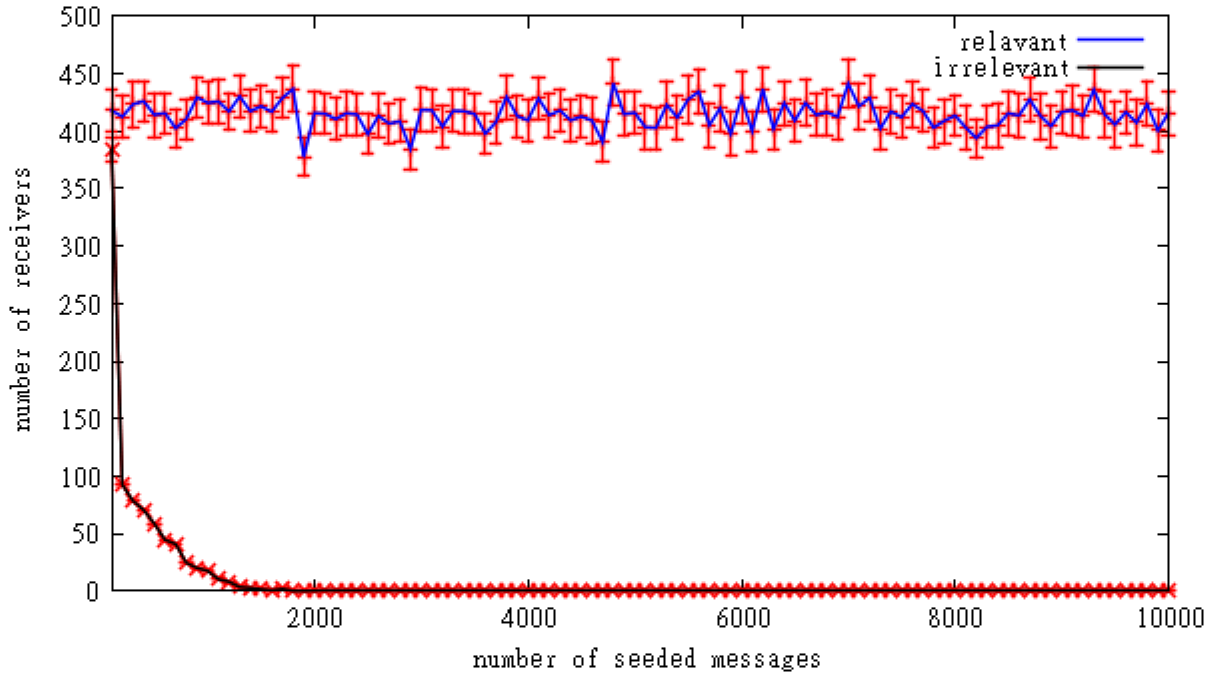
5.5.2 Experiment 2: Does Learning Rate Affect the Filtering Rate?

To explore the behaviour of the filtering mechanism with different values for the learning rate in the reinforcement equation 3.1, experiments were done on two subsets with 427 and 1300 nodes. The results are illustrated in the following Figure 5.8. In these experiments the learning rate is changed from 0.9 to 0.6 which resulted in quicker filtering of irrelevant messages by the network; it converges at around 1500 seeded messages. With this it can be concluded that the point of convergence depends on the learning rate used by the agents in updating their relationships, rather than the size of the network.

It should be noted that in our experiments all nodes used the same learning rates. In a real system, the rates can be set up by users and may be different; therefore, the point of convergence may depend on the learning rate used by most of the users. Therefore a recommendation for a system designer can be to pick a smaller default learning rate (value closer to 0.5) for a small network that is expected to generate less traffic, so that convergence can be achieved sooner.



a) 427 nodes



b) 1300 nodes

Figure 5.8: Flow of relevant vs. irrelevant messages through the network for learning rate 0.6

5.5.3 Experiment 3: How to Ensure Serendipity through Global Popularity?

In the following experiment, the popularity parameter of the message is considered before filtering irrelevant messages. Figures 5.9 to 5.14 depict the effect of considering the popularity parameter. They show the convergence of irrelevance ratio to zero, while ensuring a fairly constant serendipitance ratio. The irrelevance ratio is defined as the number of nodes for whom the forwarded message is irrelevant over the total number of nodes that received the message. The serendipitance ratio is defined as the number of node for who the forwarded message is serendipitous (not in an area of interest, but desirable due to its popularity) over the total number of nodes who received the forwarded message. Each data point is the mean of 10 runs of the experiment for the same social graph. Averages and standard deviations over 10 runs are plotted with black and blue colors respectively.

Without considering the popularity parameter, the flow of irrelevant popular information dies out. However, if the popularity parameter is considered, the flow of irrelevant popular information does not diminish and thus the nodes experience a flow of serendipitous messages throughout the run of the simulation. The popularity parameter of the social updates is constant during a run of the simulation. The seeding of popular messages (messages with a high value of their popularity parameter) is done at a regular interval of time, in this experiment 1 such message after every 20 seeded messages. In the real world popular information could be a viral video which goes around the world which becomes popular very quickly, but may belong to an interest group which is not interesting for every user.

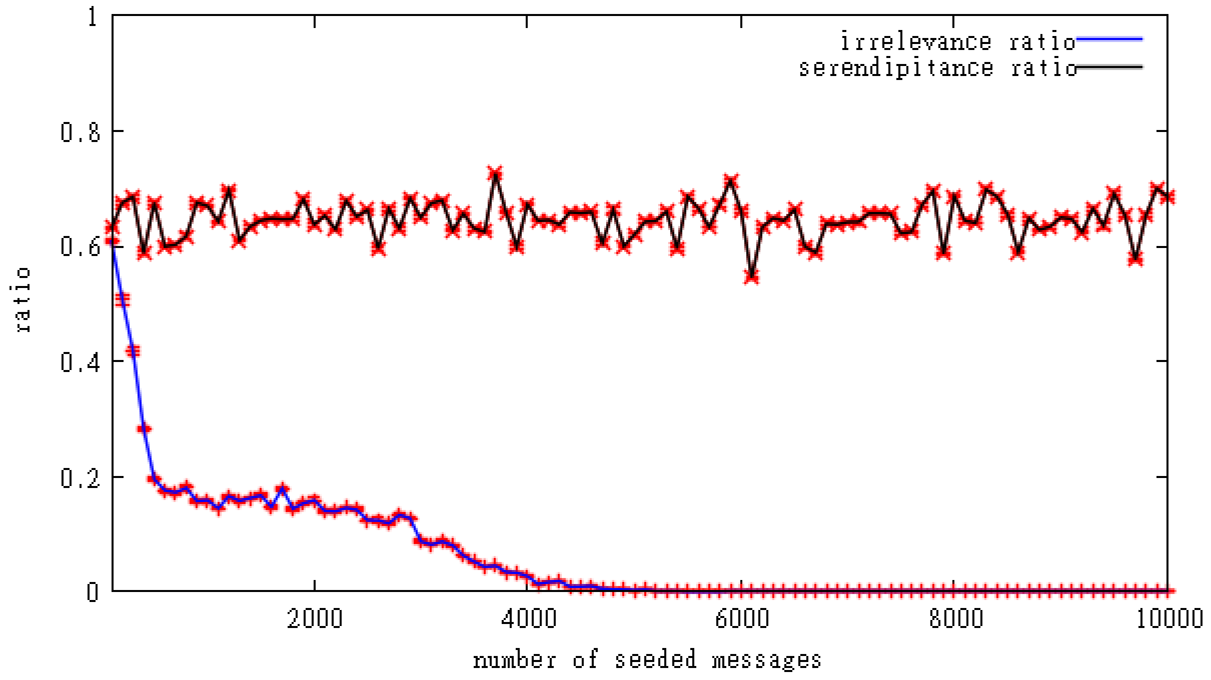


Figure 5.9: Result of Experiment 2 for subsets of 427 nodes

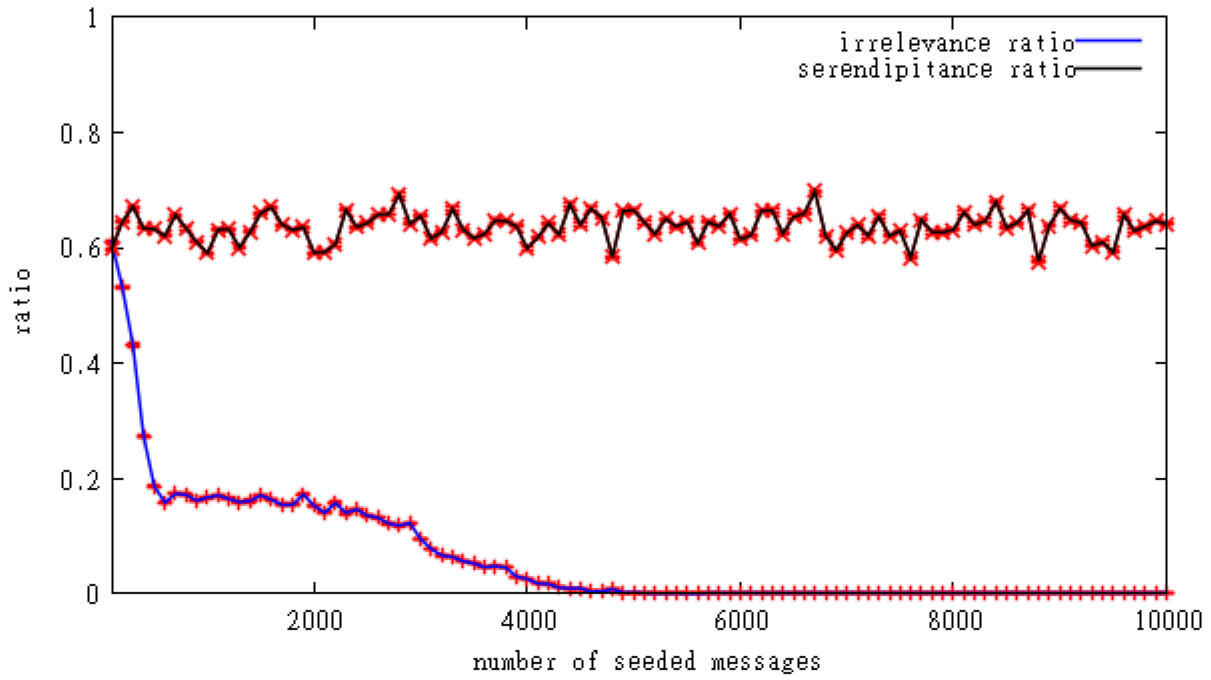


Figure 5.10: Result of Experiment 2 for subsets of 1300 nodes

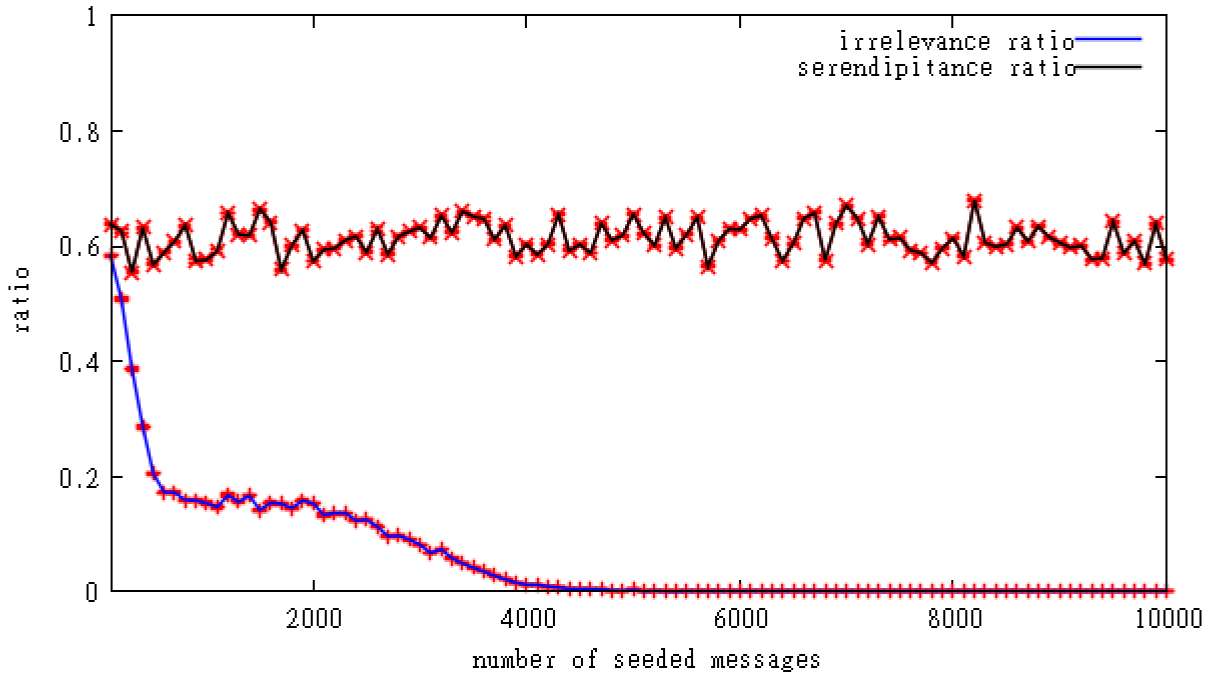


Figure 5.11: Result of Experiment 2 for subsets of 2613 nodes

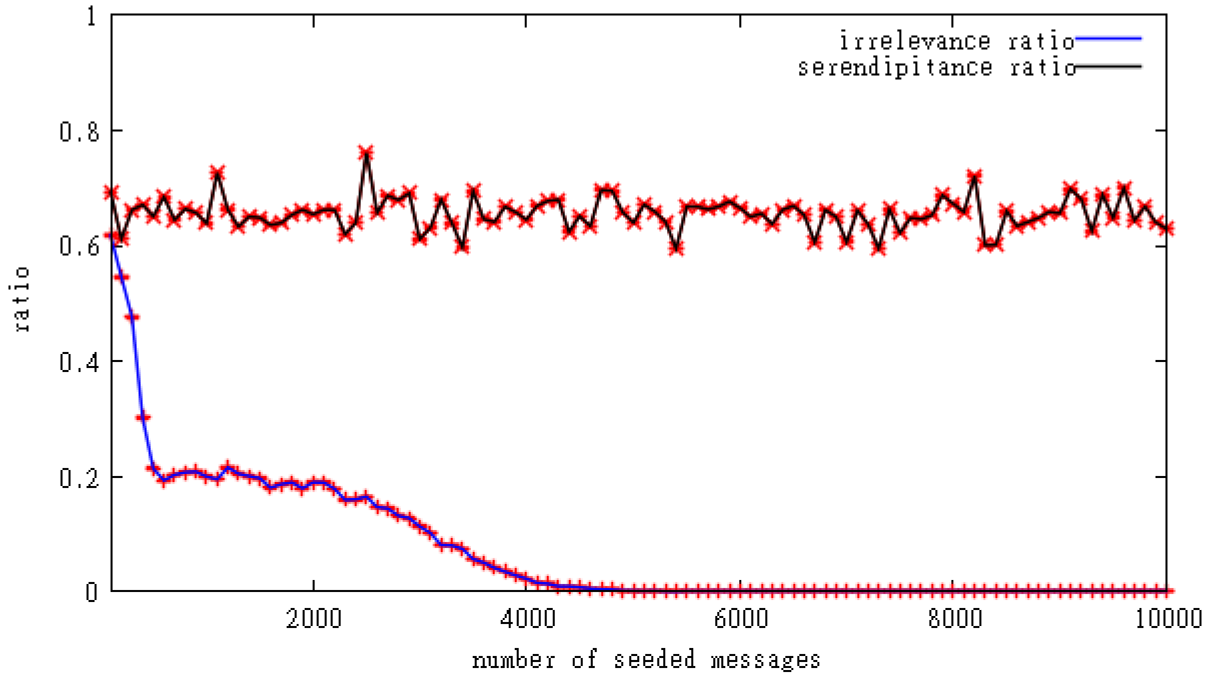


Figure 5.12: Result of Experiment 2 for subsets of 5324 nodes

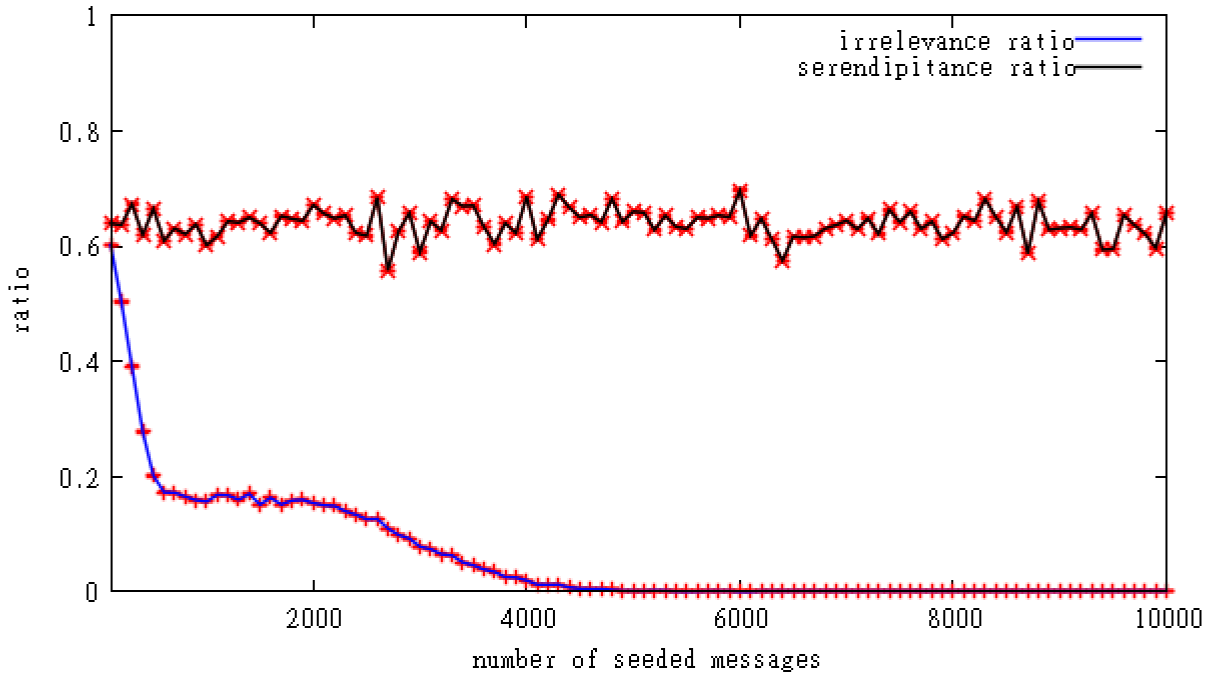


Figure 5.13: Result of Experiment 2 for subsets of 7062 nodes

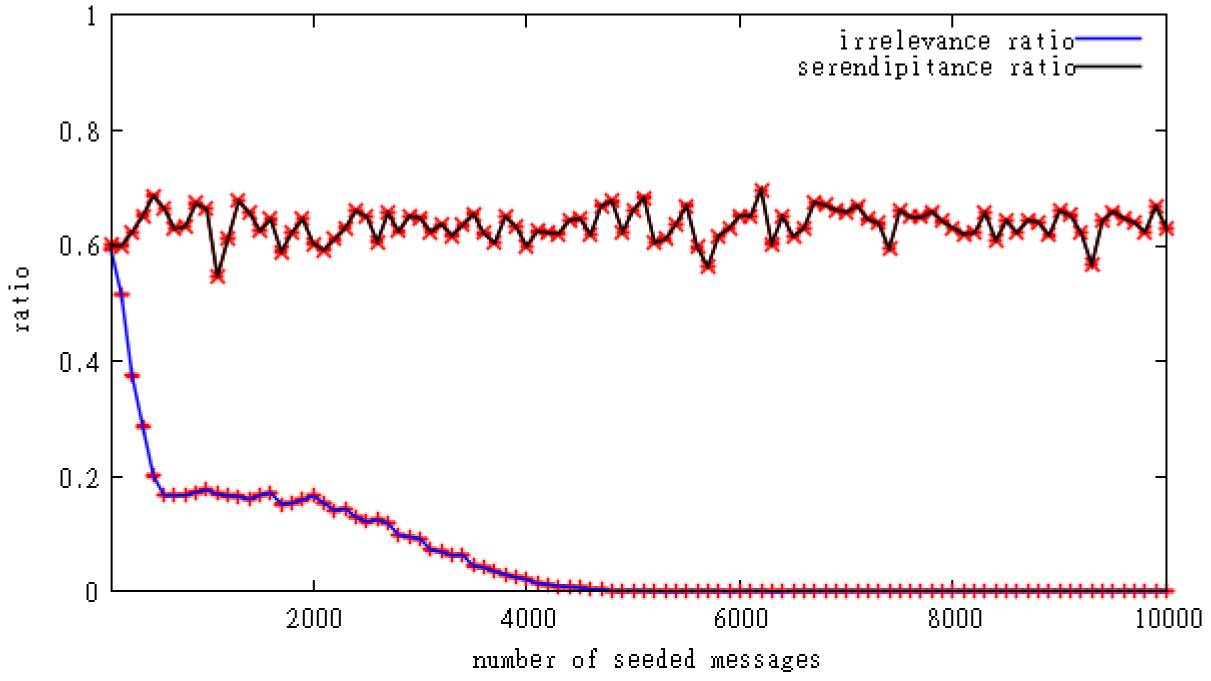


Figure 5.14: Result of Experiment 2 for subsets of 9094 nodes

5.5.4 Experiment 4: How to Ensure Serendipity through Local Popularity?

The next set of experiments is done by considering local popularity to ensure serendipitous information flow in the network. Local popularity is calculated measuring how many times the messages have been shared, liked/commented and viewed in the local community, i.e. by the user's friends. According to the calculated local popularity, the mechanism decides whether to pass the message through the filtering mechanism. Figure 5.15 and Figure 5.16 illustrate the serendipitance ratio in experiments done with 427 and 1300 nodes respectively. Each result shows three consecutive runs.

Consecutive runs on the simulation (run 1, 2, and 3, shown in the graphs with red, black and blue colours, respectively) were done by seeding the same collections of messages three times through the network. This was done to see the effect on the local popularity as it depends on the sharing, rating and viewing by friends. As shown by the results, after each run, there is an improvement in the serendipitance ratio in the next run due to the increasing popularity of certain social data. The improvement of the serendipitance ratio indicates that there has been increase in local popularity of some of the social data circulating in the system. In a real world scenario, news which is shared and viewed by most of the user's friends will be considered as desirable even when the category of the news does not interest the user.

Since the calculation of local popularity is done by querying about each friend, the simulation requires a large amount of memory, computing power and time. Therefore, in this section, results from only two experiments with lower number of nodes (427 and 1300) are illustrated. From the above two sections, it has already been shown that size of social network is independent on the rate at which the system learns to filter out irrelevant messages.

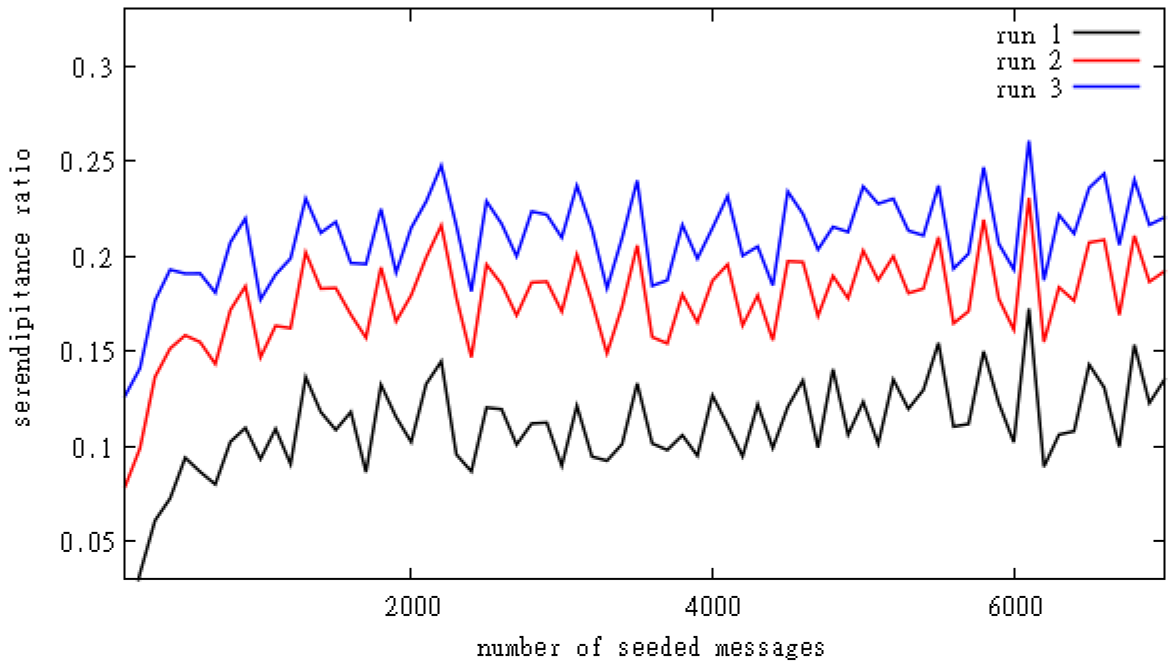


Figure 5.15: Result of Experiment 3 for subsets of 427 nodes

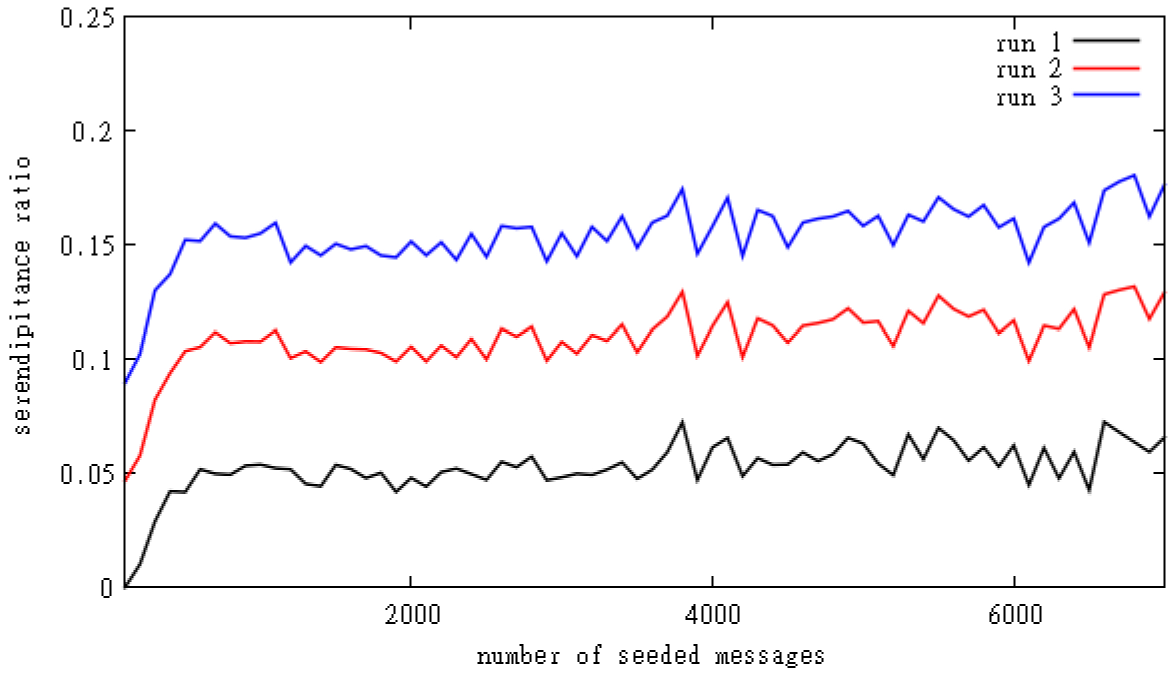


Figure 5.16: Result of Experiment 3 for subsets of 1300 nodes

5.6 Discussion

The results of the simulation as shown in the previous sections are quite positive. While the evaluation is just a proof of concept, showing that the approach is feasible and leads to the desired results, it also shows that even in a fairly sparse social network, the filtering mechanism ensures that nodes receive only information they are interested in. In addition, the mechanism can be modified to ensure that serendipitous information (based on its global or local popularity) can pass through the network, thus overcoming the "bubble effect" of traditional filters. The size of the network is not important, and the speed of convergence to relevant-data depends only on the value of the learning coefficient in the mechanism.

In comparison with other content-based filtering approaches, e.g. [1], [28], no pure interest model is created, but the interest model is an overlay over the social relationships between users. This allows more flexibility, since filtering differentiates both based on the source of information and on the user’s interest level in a category, and in addition, the relationships do not need to be symmetrical. Traditional collaborative filtering recommenders assume symmetrical relationships (even if implicit) as they use Pearson correlation between users as a strength of the relationship. In reality, however, a user A may be interested in receiving news in particular area from user B, but not the reverse, i.e. user B may not like to receive news about this area of interest from A. This allows capturing the trust that a user may have in other users’ criterions for recommending/forwarding social updates in a particular area of interest. Thus, this approach is more similar to trust-based approaches for recommending services; e.g. [42], [26] which share some similarities to this approach.

The simulation uses certain assumptions which put limitations on the validity of results. In the experiment, the number of categories of interest is fixed to 10, which is quite few in comparison to reality (e.g. if user-generated tags or Latent Semantic Analysis are used to define the areas, the number of categories can be unlimited). Adding more categories to the relationship vector may lead to sparse social graphs in certain "niche" categories, and as a result, the values of relationships in these will not be frequently updated, and will stay close to the initial value. This is actually strength of the proposed approach, as updates in these rare categories will be propagated easier through the network, leading to more serendipity. The power of the filtering mechanism will be stronger for popular categories, where more updates are propagated, and therefore the chance of getting user feedback and eventually lowering the relationship value is higher.

Another assumption is that each shared social update belongs to a single interest area, which is done for the simplicity of calculation in the simulation. In the real world, the same update can be classified into several categories of interest at the same time. In a real system, the effect will be that several relationship strengths in different categories will be updated simultaneously when the user generates feedback to a given update. While this would certainly lead to performance and scalability issues in a simulation due to the increased computation at every node, it won't be a problem in a real decentralized OSN, as each node is a separate application running either on a cloud or on a server.

The interest value of each node in a category remains static during the single run of the experiment. However, in reality users build or lose interest in different areas over time. This is not a limitation of our approach; it was just a limitation of the simulation, as it was very hard to find guidelines for simulating the dynamics of change of user interests in time. In a real system, the interest of user is expressed through his or her feedback (ratings, comments, posts) and the proposed approach allows the system to learn and adapt its relationships strengths based on this feedback.

Though the approach has not been implemented and evaluated in a real decentralized OSN, the experiments demonstrated that the relationship model reduces noise over signals and consideration of popularity in information can ensure serendipity for the nodes in the social network.

CHAPTER 6

CONCLUSION

This chapter provides a summary of the accomplished work, a list of contributions made, and outlines directions for future work.

6.1 Summary

This research has proposed an approach to deal with three problems encountered in current OSNs:

- privacy (by proposing a decentralized OSN architecture to be used),
- information overload (by proposing a novel information filtering approach based on modeling relationship strength between users in different categories), and
- avoiding filter bubbles (by excluding certain popular information from filtering).

The problem with privacy and data ownership is handled in our approach by adopting a decentralized architecture in which users host their own social data. Information overload is inevitable in Online Social Networks where users are sharing their personal data (photos, documents, status updates, etc.). Recommender systems have related works which reduces information overload but they mostly have referred to centralized architecture. Due to the adoption of a decentralized architecture, the problem with information overload in Online Social Networks persists and should be handled in decentralized way.

This thesis presents an approach of filtering technique social data in decentralized social networks, which is an alternative to centralized ones giving users control over their social data, according to the interests of users and their social relationships. The approach addresses the "filter bubble" problem by allowing serendipitous messages that are popular to pass through the filter. A model of interest in categories of social data is overlayed over a model of the strength of user interpersonal relationships to achieve filtering of information. The model is updated based on implicit and explicit feedback based on what the user does with the social data.

The evaluation of the approach is done through a simulation of the mechanism in subsets of a real social graph. The experiments showed that irrelevant information are filtered away with the help of our approach. It is also shown that change in learning rate of the reinforcement formula which is used to calculate relationship strength does change the rate at which system reduces down to zero the irrelevant information. Another set of experiments were done to evaluate how using global and local popularity of updates ensures flow of serendipitous information in our approach. In conclusion, the proposed mechanism ensured relevant and serendipitous information flow in a simulated realistic decentralized online social network.

6.2 Contributions

This thesis has following contributions:

1. A new model of relationship strength over different areas of interest was proposed to filter out irrelevant information in DOSN.
2. A method for calculating local popularity was introduced as a way to ensure the flow of serendipitous information to overcome the "filter bubble" created due to the filtering mechanism.
3. An agent based model and simulation was implemented using Erlang which is flexible, allows experimenting with different social graph topologies and agent

behaviours, and can be extended to incorporate larger experiments and other approaches.

6.3 Future Work

There are three main directions in which this work can be continued and extended.

6.3.1 Implementing the Mechanism in a Real Decentralized OSN

A natural next step would be implementing the mechanism in a real decentralized OSN and evaluating it. There are already several open-source projects which have implemented Decentralized OSN: Diaspora[15] [60], PeerSoN[16] [61] and Friendica[17] [62], to name a few. Taking advantage of the infrastructure these projects have, the proposed approach can be implemented as an extension in any one of them. The advantage of using the infrastructure of such project is to have more focus on the proposed mechanism and evaluating it. The extension can be deployed with the users of these projects and can be contributed to the community as an open source code.

6.3.2 Extending the Approach

The proposed approach could be extended by introducing multiple areas of interest associated to a social update. This would requires an algorithm to determine how much a social update is distributed over multiple areas of interest and accordingly determine the relationship strength on each areas of interest with user's friend. This would add complexity in the system but will be more realistic since in real world a social update are most often belongs to more than one category.

[15] http://diasporaproject.org/
[16] http://www.peerson.net/
[17] http://friendica.com/

There are some challenges that real implementation will face which are not included in the simulation. The determination of areas of interest a social update belongs to can be implemented either by letting users tag the social updates manually or by implementing an algorithm that automatically categorizes it, e.g. Latent Semantic Analysis (LSA). If the system implements tagging functionality, it would add an overhead for the user while sharing a social updates and there is no guarantee that user will tag the updates according to an agreed upon categorization. An alternative to tagging is using computational linguistic LSA algorithm to categorize the social updates. In this case there won't be a fixed number of categories, but potentially endless list. Yet the categories will be likely following a power-law with some (few) common categories, and many other - niche categories, would be popular only for certain small groups of users. Allowing the client to develop their own category set (by tagging or using LSA) would naturally capture diverse interest sub-groups and evolve to filter out news related to the dominant common categories.

6.3.3 Simulation Extensions

The simulation can be extended to support larger social graphs. The simulation should be implemented so that it could be distributed over multiple machines, which is well supported in Erlang. This will allow taking advantage of resources (RAM, processing power) available in multiple machines. There could be network delays and the simulation might run slower than the current version but it would support larger social graphs for evaluation.

In a real world, each person has her own rate at which she learns which topic is interesting and which friend sends most relevant information to her in the network. Current version of the simulation has this learning rate same throughout the population in the network. It can be extended to incorporate learning rate distribution over the population following power law distribution. With this, the simulation can come close to the real world where learning rate of people is different from each other.

We also want to explore how the model can predict certain group phenomena, by varying certain parameters and what parameter values (e.g. learning rate, thresholds) should be recommended for particular sizes and types of networks. For example, increase and decrease in the values associated with the calculation of relationship strength will change the rate of learning of the network about irrelevant social data.

LIST OF REFERENCES

1. Abel, F., Gao, Q., Houben, G.J., and Tao, K. Analyzing User Modeling on Twitter for Personalized News Recommendations. *International Conference on User Modeling, Adaption and Personalization (UMAP), Girona, Spain*, (2011).

2. Adar, E. and Huberman, B.A. Free riding on gnutella. *First Monday 5*, 10 (2000), 2–13.

3. Anwar, Z., Yurcik, W., Pandey, V., Shankar, A., Gupta, I., and Campbell, R.H. Leveraging Social-Network Infrastructure to Improve Peer-to-Peer Overlay Performance: Results from Orkut. *ACM Computing Research Repository*, (2005).

4. Armbrust, M., Fox, A., Griffith, R., et al. A view of cloud computing. *Commun. ACM 53*, 4 (2010), 50–58.

5. Armstrong, J. Erlang - A survey of the language and its industrial applications. *The 9'th Exhibitions and Symposium on Industrial Applications of Prolog*, (1996).

6. Backstrom, L. Anatomy of Facebook. 2011. https://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859.

7. Berners-Lee, T., Hollenbach, J., Lu, K., Presbrey, J., and Schraefel, M. Tabulator redux: Browsing and writing linked data. *Linked Data on the Web Workshop at WWW08*, (2008).

8. Berners-Lee, T. Linked Data. 2006. http://www.w3.org/DesignIssues/LinkedData.html.

9. Billsus, D. and Pazzani, M.J. A hybrid user model for news story classification. *Proceeding of the 7th International Conference on User Modeling (UM '99)*, (1999), 99–108.

10. Bonabeau, E. Agent-based modeling: methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences of the United States of America*, (2002), 7280–7287.

11. Boyd, D. and Ellison, N.B. Social Network Sites: Definition, History, and Scholarship. *Journal of Computer-Mediated Communication 13*, 1 (2008), 210–230.

12. Brickley, D. and Miller, L. FOAF Vocabulary Specification 0.98. 2010. http://xmlns.com/foaf/spec/.

13. Buchegger, S., Schiöberg, D., Vu, L.H., and Datta, A. PeerSoN: P2P social networking: early experiences and insights. *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, ACM (2009), 46–52.

14. Buford, J.F. and Yu, H. Peer-to-Peer Networking and Applications: Synopsis and Research Directions. In X. Shen, H. Yu, J. Buford and M. Akon, eds., *Handbook of Peer-to-Peer Networking*. Springer, 2010.

15. Bullas, J. 6 Social Media Networks to Watch in 2012 [Plus Infographics]. 2012. http://www.jeffbullas.com/2012/01/03/6-social-media-networks-to-watch-in-2012-plus-infographics/.

16. Chen, L. and Sycara, K. WebMate: a personal agent for browsing and searching. *Proceeding of the 2nd international conference on autonomous agents and multi agent systems*, (1998), 132–139.

17. Cozzatti, J.-P. A Perfect Storm of Whales. 2010. http://engineering.twitter.com/2010/06/perfect-stormof-whales.html.

18. Datta, A., Buchegger, S., Vu, L.H., Strufe, T., and Rzadca, K. Decentralized Online Social Networks. In B. Furht, ed., *Handbook of Social Network Technologies and Applications*. Springer, 2010, 349–378.

19. Datta, A. and Sharma, R. GoDisco: Selective Gossip Based Dissemination of Information in Social Community Based Overlays. *Distributed Computing and Networking*, (2011), 227–238.

20. Fritsch, H. StudiVZ - Inoffizielle Statistiken vom Dezember 2006. http://studivz.irgendwo.org/.

21. Heckmann, D., Schwartz, T., and Brandherm, B. Decentralized user modeling with UserML and GUMO. *Workshop on Decentralized, Agent Based and Social Approaches to User Modelling (DASUM), 9th Intl Conference on User Modeling, Edinburgh, Scotland*, (2005), 61–64.

22. Iaquinta, L., Gemmis, M. De, Lops, P., Semeraro, G., Filannino, M., and Molino, P. Introducing Serendipity in a Content-Based Recommender System. *2008 Eighth International Conference on Hybrid Intelligent Systems*, Ieee (2008), 168–173.

23. Koubarakis, M., Tryfonopoulos, C., Idreos, S., and Drougas, Y. Selective Information Dissemination in P2P Networks: Problems and Solutions. *SIGMOD Rec. 32*, 3 (2003), 71–76.

24. Murakami, T., Mori, K., and Orihara, R. Metrics for Evaluating the Serendipity of. In *New Frontiers in Artificial Intelligence*. 2008, 40–46.

25. Narayanan, A. and Shmatikov, V. Robust de-anonymization of large sparse datasets. *2008 IEEE Symposium on Security and Privacy (sp 2008)*, Ieee (2008), 111–125.

26. Nusrat, S. and Vassileva, J. Recommending Services in a Trust-Based Decentralized User Modeling System. In L. Ardissono and T. Kufik, eds., *Advances in User Modeling: selected papers from UMAP 2011 workshops*. Springer Berlin Heidelberg, 2011.

27. Pariser, E. *The Filter Bubble: What the Internet is Hiding From You.* Penguin Press HC, New York, 2011.

28. Pazzani, M.J. and Billsus, D. Content-based recommendation systems. *The adaptive web*, (2007), 325–341.

29. Perez, J.C. Facebook's Beacon More Intrusive Than Previously Thought. 2007. http://www.pcworld.com/article/140182/facebooks_beacon_more_intrusive_than_previou sly_thought.html.

30. Pouwelse, J.A., Garbacki, P., Wang, J., et al. TRIBLER: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience 20*, 2 (2008), 127–138.

31. Ramakrishnan, N., Keller, B.J., Mirza, B.J., Grama, A.Y., and Karypis, G. Privacy risks in recommender systems. *IEEE Internet Computing 5*, 6 (2001), 54–62.

32. Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ACM (2001), 161–172.

33. Resnick, P., Iacovou, N., and Suchak, M. Grouplens: An open architecture for collaborative filtering of netnews. *Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work*, (1994), 175–186.

34. Resnick, P. and Varian, H.R. Recommender Systems. *Communications of the ACM 40*, 3 (1997), 56–58.

35. Rhea, S., Godfrey, B., Karp, B., et al. OpenDHT: A public DHT service and its uses. *ACM SIGCOMM Computer Communication Review 35*, 4 (2005), 73–84.

36. Rich, E. User modeling via stereotypes. *Cognitive science 3*, 4 (1979), 329–354.

37. Rich, E. Users are individuals: individualizing user models. *International journal of man-machine studies*, May 1981 (1983), 199–214.

38. Seaborne, A., Manjunath, G., Bizer, C., et al. SPARQL Update. 2008. http://www.w3.org/Submission/SPARQL-Update/.

39. Sun, L., Upadrashta, Y., and Vassileva, J. Ensuring quality of service in p2p file sharing through user and relationship modelling. *Proceedings of the User Modelling UM03 Workshop on User and Group Models for Web-Based Adaptive Collaborative Environments, Johnstown*, (2003), 57–66.

40. Tandukar, U. and Vassileva, J. Selective Propagation of Social Data in Decentralized Online Social Network. In *Advances in User Modeling: selected papers from UMAP 2011 workshops*. Springer, 2011.

41. Tandukar, U. and Vassileva, J. Ensuring Relevant and Serendipitous Information Flow in Decentralized Online Social Network. *Artificial Intelligence: Methodology, Systems, and Applications. 15th International Conference, AIMSA 2012*, (2012), 79–88.

42. Wang, Y. and Vassileva, J. Trust and reputation model in peer-to-peer networks. *Proceedings Third International Conference on Peer-to-Peer Computing (P2P2003)*, September (2003), 150–157.

43. Webster, A. and Vassileva, J. Push-poll recommender system: Supporting word of mouth. *LNCS Proceedings User Modelling, UM2007, Corfu, Greece*, Springer (2007), 278–287.

44. Webster, A. and Vassileva, J. The keepup recommender system. *Proceedings of the 2007 ACM conference on Recommender systems*, ACM (2007), 173–176.

45. Whitehead Jr, E.J. and Goland, Y.Y. WebDAV: A network protocol for remote collaborative authoring on the Web. *Proceedings of the sixth conference on European Conference on Computer Supported Cooperative Work*, Kluwer Academic Publishers (1999), 291–310.

46. Wikipedia. List of social networking websites. *Wikipedia*. http://en.wikipedia.org/wiki/List_of_social_networking_websites.

47. Wilson, C., Boe, B., Sala, A., Puttaswamy, K.P.N., and Zhao, B.Y. User interactions in social networks and their implications. *Proceedings of the 4th ACM European conference on Computer systems*, Acm (2009), 205–218.

48. Yeung, C.A., Liccardi, I., Lu, K., Seneviratne, O., and Berners-Lee, T. Decentralization: The future of online social networking. *W3C Workshop on the Future of Social Networking Position Papers*, (2009).

49. Alternative Interfaces. http://www.imdb.com/interfaces.

50. SwarmWiki. http://www.swarm.org/.

51. The Repast Suite. http://repast.sourceforge.net/.

52. MASON Multiagent Simulation Toolkit. http://cs.gmu.edu/~eclab/projects/mason/.

53. NetLogo. http://ccl.northwestern.edu/netlogo/.

54. Anylogic Simulation Software. http://www.xjtek.com/.

55. Erlang Programming Language. http://www.erlang.org/.

56. Python Programming Language – Official Website. http://www.python.org/.

57. Overview -- NetworkX 1.7 documentation. http://networkx.lanl.gov/.

58. gnuplot homepage. http://www.gnuplot.info/.

59. Networks / Pajek. http://vlado.fmf.uni-lj.si/pub/networks/pajek/.

60. The Diaspora Project. http://diasporaproject.org/.

61. PeerSoN: Privacy-Preserving P2P Social Networks. http://www.peerson.net/.

62. The internet is our social network. http://friendica.com/.

APPENDIX A
DATASET PARSER AND SUBSET CREATOR WRITTEN IN PYTHON

1. Dataset reader written in python

```
## Reading from StudiVZ friendlist dataset and putting them in dictionary
## key of dictionary is the user node and value is list of her friends
def parse_graph(filename):
    f_socialgraph = open(filename,"r")
    iteration = 0
    graph = dict()
    for line in f_socialgraph:
        if len(line)>0:
            fields = line.split()
            if len(fields)>0:
                uid = fields[0]
                check = uid in graph
                if check == False:
                    graph[uid] = list()
                for i in range(len(fields)):
                    if i != 0:
                        graph[uid].append(fields[i])
        iteration = iteration + 1
        if (iteration % 10**5== 0 ):
            sys.stdout.write('.')
            sys.stdout.flush()
    f_socialgraph.close()
    return graph
##########################################################################
```

2. Code snippet creating subset of graph

```
## Getting the subset of social graph from the selected hubs
default = list()
my_social_graph = dict()
active = my_hub
visited = list()
iteration = 0
while 1:
    if len(active) <= 0:
        break
    print(len(active))
    iteration = iteration + 1

    node = active.pop(0)
    if any(element == node for element in visited) == False:
        node_friends = graph.get(node,default)
        if (node in my_social_graph) == False:
            my_social_graph[node] = list()
        my_social_graph[node] = list(set(my_social_graph[node]) |
                                     set(node_friends))
        visited.append(node)
        for e in node_friends:
            if (e in my_social_graph) == False:
                my_social_graph[e] = list()
            my_social_graph[e].append(node)

        if len(node_friends) > 0:
            intersect = list(set(node_friends) & set(visited))
            per_of_connectivity = (len(intersect)/len(node_friends))*100
        else:
            per_of_connectivity = 0

        if len(node_friends) > 0 and (len(node_friends) < no_of_fof or
                                      len(visited) < no_of_hubs or
                                      per_of_connectivity > 25):
            active = active + node_friends
return my_social_graph
```

3. Code snippet used to write subset graph to a file in tab-separated value (tsv) format and function returning properties of graph like no. of nodes, edges, clustering coefficient, etc.

```python
## Writing social graph in tsv format
def create_tsv_format(folder_name,graph):
    f = open(folder_name+"/social-graph.tsv","w")
    i = 0
    for key in graph:
        f.write('%s\t' % (key))
        for i in range(len(graph[key])):
            f.write('%s\t' % (graph[key][i]))
        f.write('\r\n')
    f.close()
###########################################################################

## Creating NetworkX graph for the analysis of the sub-graph
def network_analysis(graph):
    statistics = dict()
    G = nx.Graph()
    total_friends = 0
    for key in graph:
        G.add_node(key)
    for key in graph:
        total_friends += len(graph[key])
        for i in range(len(graph[key])):
            G.add_edge(key,graph[key][i])

    statistics["number_of_nodes"] = G.number_of_nodes()
    statistics["number_of_edges"] = G.number_of_edges()
    statistics["clustering_coefficient"] = nx.average_clustering(G)
    statistics["transitivity"] = nx.transitivity(G)
    statistics["average_friends"] = total_friends / G.number_of_nodes()
    return statistics
###########################################################################
```

4. Code snippet used distribute interest over the population in the given social graph

```
## Generating interest distribution
def distribute_interests(graph,no_of_category):
    interests = dict()
    for i in range(0,no_of_category):
        interests[i] = 1/((i+2)**0.45)   #formula for the distribution
    #--> 0.45 is the parameter that determines the shape of the curve

    d = dict()
    graph_length = len(graph)
    #print(interests)
    for x in range(len(interests)):
        popular_among = int(math.ceil(interests[x] * graph_length))
        d[x] = [0.3] * graph_length
        visited = dict()
        i = 0
        for z in range(0,popular_among):
            while(-1):
                rand = random.randint(0,graph_length-1)
                if(rand in visited):
                    continue
                else:
                    d[x][rand] = float(random.randint(70,100))/float(100)
                    visited[i] = rand
                    #print(rand)
                    i += 1
                    break


    interest_distribution = dict()
    i = 0
    for key in graph:
        interest_distribution[key] = dict()
        for x in range(0,no_of_category):
            interest_distribution[key][x] = d[x][i]
        i += 1

    return interest_distribution
```

APPENDIX B
SIMULATOR WRITTEN IN ERLANG

1. Agent module which is responsible for functionality of agent process.

```
%%
%% This is to create a new node along with the list of its friends
%%
create(Node,Name,FriendList,CatList) ->
    Interests = getInterest(Name,CatList),
    Relations = getRelationship(Name,FriendList,Interests,[]),

    MsgTypeTotal = {0,0,0,0,0,0,0,0},

    global:register_name(Name,
                         spawn(Node,
                               ?MODULE,
                               loop,
                               [Name,FriendList,Interests,
                                Relations,[],MsgTypeTotal])).

%%
%% For bootstrap setting the interest level in the node
%%
getInterest(Name,CatList) ->
    List = getInterestList(Name,CatList,[]),
    List.

getInterestList(_,[],Acc)->
    Acc;
getInterestList(Name,CatList,Acc) ->
    [CatName | RemList] = CatList,
    Value = 1.0,
    NewAcc = Acc ++ [{CatName,Value}],
    getInterestList(Name,RemList,NewAcc).

%%
%% For bootstrap getting relationship model for each friend/neighbouring nodes
%%
getRelationship(_Name,[],_Interests,Model) ->
    Model;
getRelationship(Name,FriendList,Interests,Model) ->
    [F | Rest] = FriendList,
    FriendModel = getModel(Name,F,Interests,[]),
    Item = {F,FriendModel},
    NewModel = [Item | Model],
    getRelationship(Name,Rest,Interests,NewModel).
```

```
%%
%% Setting up the relationship model for a friend in the beginning of the system
%%
getModel(_Name,_Friend,[],Model)->
    Model;
getModel(Name,Friend,Interests,Model) ->
    [F | Rest] = Interests,
    {Cat,_} = F,
    Strength = 1.0,
    Item = {Cat,Strength},
    NewModel = [Item | Model],
    getModel(Name,Friend,Rest,NewModel).

%%
%% This function is responsible for listening to incoming messages and
%% responding accordingly
%%
loop(Name,FriendList,Interests,Relations,MsgList,MsgTypeTotal) ->
    receive
        {seed,Message} ->
            global:whereis_name(monitor) ! {alive},
            {MsgID,_Text,Category,_Hops,_Popularity} = Message,
            NewMsgList = [MsgID | MsgList],
            {Seeded,NonPopularRelevant,NonPopularIrrelevant,
             PopularRelevant,PopularIrrelevant,
             Filtered,UniqueMsg,TotalMsg} = MsgTypeTotal,
            NewMsgTypeTotal = {Seeded+1,NonPopularRelevant,NonPopularIrrelevant,
                               PopularRelevant,PopularIrrelevant,Filtered,
                               UniqueMsg+1,TotalMsg+1},
            TotalForwards = forwardMessage(Message,Name,FriendList,0),

            loop(Name,FriendList,Interests,Relations,NewMsgList,NewMsgTypeTotal);

        {forward,Message,From} ->
            global:whereis_name(monitor) ! {alive},
            {MsgID,_Text,Category,Hops,Popularity} = Message,
            HasMsg = lists:member(MsgID, MsgList),

            %% add this message to the message list
            NewMsgList = [MsgID | MsgList],

            %% find the relationship strength between the friend on the category
            %% of the message
            RelTuple = lists:keyfind(From,1,Relations),
            {_,Rel} = RelTuple,
            CategoryTuple = lists:keyfind(Category,1,Rel),
            {_,RelVal} = CategoryTuple,

            InterestTuple = lists:keyfind(Category,1,Interests),
            {_,IntVal} = InterestTuple,

            %% analyzing the message by the reciever
            Threshold = analyzeMessage(Interests,Message),

        MsgName = list_to_atom("msg_" ++ integer_to_list(MsgID)),
        global:whereis_name(MsgName) ! {forward,
                                        {From,Name,Hops,IntVal,Threshold,RelVal}},
{Seeded,NonPopularRelevant,NonPopularIrrelevant,PopularRelevant,
 PopularIrrelevant,Filtered,UniqueMsg,TotalMsg} = MsgTypeTotal,

%% for analysis of data passing through the node
if
    Popularity >= 0.9, IntVal >= 0.5 ->
        NewNonPopularRelevant = NonPopularRelevant,
        NewNonPopularIrrelevant = NonPopularIrrelevant,
        NewPopularRelevant = PopularRelevant + 1,
        NewPopularIrrelevant = PopularIrrelevant;
    Popularity >= 0.9, IntVal < 0.5 ->
        NewNonPopularRelevant = NonPopularRelevant,
        NewNonPopularIrrelevant = NonPopularIrrelevant,
        NewPopularRelevant = PopularRelevant,
        NewPopularIrrelevant = PopularIrrelevant + 1;
    Popularity < 0.9, IntVal >= 0.5 ->
        NewNonPopularRelevant = NonPopularRelevant + 1,
        NewNonPopularIrrelevant = NonPopularIrrelevant,
        NewPopularRelevant = PopularRelevant,
        NewPopularIrrelevant = PopularIrrelevant;
    Popularity < 0.9, IntVal < 0.5 ->
        NewNonPopularRelevant = NonPopularRelevant,
        NewNonPopularIrrelevant = NonPopularIrrelevant + 1,
        NewPopularRelevant = PopularRelevant,
        NewPopularIrrelevant = PopularIrrelevant;
    true ->
        NewNonPopularRelevant = NonPopularRelevant,
        NewNonPopularIrrelevant = NonPopularIrrelevant,
        NewPopularRelevant = PopularRelevant,
        NewPopularIrrelevant = PopularIrrelevant
end,
if
    HasMsg == false ->
        NewUniqueMsg = UniqueMsg + 1;
    true ->
        NewUniqueMsg = UniqueMsg
end,


%% if relationship strength is more than or equal to 0.4 then
%% only receiver can view the message
if
    RelVal >= 0.4 ->
        %% according to the analysis of the message update the
        %% realtionship model
        NewRelations = updateModel(Name,From,Message,
                                   Threshold,Relations),

        %% receiver forwards the message if it is likely to be shared
        %% if it has not shared before
        if
            Threshold >= 0.9,HasMsg == false->
                TotalForwards = forwardMessage(Message,Name,
                                               FriendList,0);
            true ->
                TotalForwards = 0
        end,
                    NewFiltered = Filtered;

                true ->
                    NewFiltered = Filtered + 1,
                    NewRelations = Relations
            end,

            NewMsgTypeTotal = {Seeded,NewNonPopularRelevant,NewNonPopularIrrelevant,
                               NewPopularRelevant,NewPopularIrrelevant,NewFiltered,
                               NewUniqueMsg,TotalMsg+1},

            loop(Name,FriendList,Interests,NewRelations,NewMsgList,NewMsgTypeTotal);

        {interest,Popularity} ->
            NewInterest = distribute_interest(Name,Interests,Popularity,[]),
            loop(Name,FriendList,NewInterest,Relations,MsgList,MsgTypeTotal);

        {analyze,Pid} ->
            Pid ! {value, MsgTypeTotal},
            loop(Name,FriendList,Interests,Relations,MsgList,MsgTypeTotal)
    end.

distribute_interest(_Name,[],_Popularity,Acc) ->
    Acc;
distribute_interest(Name,InterestList,Popularity,Acc) ->
    [F | Rest] = InterestList,
    {Cat,_} = F,
    A = lists:keyfind(Cat,1,Popularity),
    {_,PopList} = A,

    Bool = lists:member(Name, PopList),

    case Bool of
        true ->
            Value = (random:uniform(51) + (49))/100;
        false ->
            Value = (random:uniform(31) + (19))/100
    end,

    NewAcc = Acc ++ [{Cat,Value}],
    distribute_interest(Name,Rest,Popularity,NewAcc).

%%
%% Message is a tuple of {MessageID,MessageText,Category,Hops}
%%
forwardMessage(_Message,_Name,[],TotalForwards) ->
    TotalForwards;
forwardMessage(Message,Name,FriendList,TotalForwards) ->
    {MsgID,Text,Category,Hops,Popularity} = Message,
    [Friend|Rest] = FriendList,
    NewMessage = {MsgID,Text,Category,Hops+1,Popularity},
    NewTotalForwards = TotalForwards + 1,
    global:whereis_name(Friend) ! {forward,NewMessage,Name},
    forwardMessage(Message,Name,Rest,NewTotalForwards).
```

```
%%
%% To analyze the message
%% We will check the category of message and check the interest
%% level of that category. Then the feedback is returned.
%%
analyzeMessage(Interests,Message) ->
    {_MsgID,_Text,Category,_Hops,_Popularity} = Message,
    InterestTuple = lists:keyfind(Category,1,Interests),
    {_,IntVal} = InterestTuple,

    if
        IntVal >= 0.9 ->
            RetVal = 0.9;
        IntVal >= 0.7,IntVal < 0.9 ->
            RetVal = 0.7;
        IntVal >= 0.5,IntVal < 0.7 ->
            RetVal = 0.5;
        true ->
            RetVal = 0.3
    end,

    RetVal.

%%
%% To update the model of relationship in a node
%% ToDo: Store to a database for analysis of evolution of system
%%
updateModel(Name,FriendName,Message,Value,Relations) ->
    {MsgID,_Text,Category,_Hops,_Popularity} = Message,

    RelTuple = lists:keyfind(FriendName,1,Relations),
    {_,Rel} = RelTuple,
    CategoryTuple = lists:keyfind(Category,1,Rel),
    {_,CatVal} = CategoryTuple,

    I = 0.9,
    %% S(X) = i * Sp(X) + (1-i) * Feedback
    NewVal = (I * CatVal) + ((1 - I) * Value),
    TmpRel = lists:delete(CategoryTuple,Rel),
    NewRel = [{Category,NewVal}|TmpRel],
    TmpMod = lists:delete(RelTuple,Relations),
    NewMod = [{FriendName,NewRel} | TmpMod],

    NewMod.
```

2. Network module responsible for creation of social graph and seeding message into the network.

```
start() ->
    db:start(),
    T = erlang:now(),
    random:seed(T),

    %% getting configuration parameters
    %% NumOfCat = number of category
    %% Filename = name of the file which has social graph in tsv format
    %% NumOfMsg = number of messages to be seeded
    %% PerToSeed = percentage of population to be seeded
    %% NumOfMsgMac = number of erlang
    {_atomic,
     [{_Input, _ID, NumOfCat, Filename, NumOfMsg,
       PerToSeed,_ErlNodes,NumOfMsgMac,NumOfSocMac}]} = db:get_input(),

    CatList = create_category_list(list_to_integer(NumOfCat), []),

    case file:open(Filename,[read]) of
        {ok,FileId} ->
            %io:fwrite(FileId, "~s~n", [Data]),
            %%Machines = string:tokens(ErlNodes, ","),
            %%ping_machines(Machines),

            Soc = list_to_integer(NumOfSocMac),
            if
                Soc == 0 ->
                    SocMachines = [node()];
                true ->
                    SocMachines = create_soc_machines(Soc,[])
            end,

            Msg = list_to_integer(NumOfMsgMac),
            if
                Msg == 0 ->
                    MsgMachines = [node()];
                true ->
                    MsgMachines = create_msg_machines(Msg,[])
            end,

            PeerList = parse_tsv(FileId,[],SocMachines,CatList,0),
            Popularity = distribute_interest(PeerList,CatList,1,[]),

            io:format("Interest distribution to nodes starting!!!~n"),
            set_interest(PeerList,Popularity,1),
            file:close(FileId),

            global:register_name(rerun,
                                 spawn(network,run,
                                       [list_to_integer(NumOfMsg),
                                        list_to_integer(PerToSeed),
                                        CatList, PeerList, Popularity,
                                        list_to_integer(NumOfMsgMac)])),
            global:register_name(storage,
                                 spawn(network,store,
                                       [PeerList,list_to_integer(NumOfMsg)])),

            global:register_name(monitor,
                                 spawn(network,process_monitor,[])),

            io:format("Seeding messages starting!!!~n"),
            spawn(network,loop,[1,list_to_integer(NumOfMsg),
                                list_to_integer(PerToSeed), CatList,
                                PeerList, Popularity, MsgMachines]);

        {error,Reason} ->
            io:format("~p",[Reason])
    end.

%% to seed more messages into the experiment again after seeding
%% certain number of messages
run(MsgNum,PerToSeed,CatList,PeerList,Popularity,NumOfMsgMac) ->
    receive
        {run,Number} ->
            Num = Number,
            spawn(network,
                  loop,
                  [MsgNum+1, MsgNum+Number, PerToSeed, CatList,
                   PeerList, Popularity,NumOfMsgMac]);
        _ ->
            Num = 0,
            true
    end,
    run(MsgNum+Num,PerToSeed,CatList,PeerList,Popularity,NumOfMsgMac).

%% pinging machines to connect
ping_machines([]) ->
    true;
ping_machines(Machines) ->
    [F|Rest] = Machines,
    net_adm:ping(list_to_atom(F)),
    ping_machines(Rest).

%% creating social graph machines to distribute node processes
create_soc_machines(0,Acc) ->
    Acc;
create_soc_machines(Num,Acc) ->
    [_,Host] = string:tokens(atom_to_list(node()),"@"),
    io:format("Creating social machine: ~p~n",[Num]),
    slave:start(list_to_atom(Host),
                list_to_atom("soc_machine_" ++
                                 integer_to_list(Num))),

    NodeName = list_to_atom("soc_machine_" ++
                                integer_to_list(Num) ++ "@" ++ Host),
    NewAcc = [NodeName | Acc],

    create_soc_machines(Num-1,NewAcc).
```

```
%% creating slave machines to distribute message processes
create_msg_machines(0,Acc) ->
    Acc;
create_msg_machines(Num,Acc) ->
    [_,Host] = string:tokens(atom_to_list(node()),"@"),
    io:format("Creating message machine: ~p~n",[Num]),
    slave:start(list_to_atom(Host),
                list_to_atom("msg_machine_" ++ integer_to_list(Num))),
    NodeName = list_to_atom("msg_machine_" ++
                                integer_to_list(Num) ++ "@" ++ Host),
    NewAcc = [NodeName | Acc],
    create_msg_machines(Num-1,NewAcc).

%% to monitor the processes --> not in use
process_monitor() ->
    receive
        {alive} ->
            process_monitor()
    after 30000 ->
            io:format("Simulation ended!!!"),
            csv:get_nodes_info(),
            csv:get_messages_info()
    end.

%% to create list of categories
create_category_list(0, Acc) ->
    Acc;
create_category_list(Num, Acc) ->
    CatName = list_to_atom("c" ++ integer_to_list(Num)),
    create_category_list(Num-1, [CatName | Acc]).

parse_tsv(FileId,PeerList,Machines,CatList,Count) ->
    Line = file:read_line(FileId),
    case Line of
        eof ->
            PeerList;
        {_,Str} = Line ->
            Node = init_peer(string:tokens(Str, "\t\n"),Machines,CatList,Count),
            parse_tsv(FileId,[Node |PeerList],Machines,CatList,Count+1)
    end.

init_peer(List,Machines,CatList,Count) ->
    [F|L] = List,
    Node = list_to_atom("n" ++ F),
    FriendList = lists:foldr(fun(X,Acc) ->
                                [list_to_atom("n" ++ X) | Acc] end,
                            [], L),
    I = (Count+1) rem length(Machines),
    peer:create(lists:nth(I+1, Machines),Node,FriendList,CatList),
    Node.

%% to distribute interest among the population
distribute_interest(_PeerList,[],_Loop,Acc)->
    Acc;
distribute_interest(PeerList,CatList,Loop,Acc) ->
    [CatName|Rest] = CatList,
    Ran = random:uniform(21) + 79,
    A = length(PeerList) * (Ran/100),
    B = round(A/math:pow(2, Loop-1)),

    if
        B < (length(PeerList) * 0.1) ->
            Ran1 = random:uniform(11) + 4,
            P = round(length(PeerList) * (Ran1/100));
        true ->
            P = B
    end,

    Res = distribute_popularity(PeerList,CatName,P,[]),
    NewAcc = Acc ++ [{CatName,Res}],
    distribute_interest(PeerList,Rest,Loop+1,NewAcc).

distribute_popularity(_PeerList,_CatName,0,Acc) ->
    Acc;
distribute_popularity(PeerList,CatName,Num,Acc) ->
    Index = random:uniform(length(PeerList)),
    Element = lists:nth(Index, PeerList),
    NewAcc = Acc ++ [Element],
    NewList = lists:delete(Element, PeerList),
    distribute_popularity(NewList, CatName, Num-1, NewAcc).

set_interest([],_Popularity,_) ->
    io:format("Interest distribution to nodes completed!!!~n"),
    true;
set_interest(List,Popularity,Loop) ->
    [Element|RestList] = List,
    global:whereis_name(Element) ! {interest,Popularity},
    if
        Loop rem 1000 == 0 ->
            io:format("*");
        true ->
            true
    end,
    %%timer:sleep(200),
    set_interest(RestList,Popularity,Loop+1).

%% loop to seed messages into the network
loop(ID, TotalMsg, PerToSeed, CatList, PeerList, Popularity, Machines) ->
    if
        ID == TotalMsg+1 ->
            global:whereis_name(storage) ! {change_num_of_msg, TotalMsg},
            io:format("~nSeeding ~p messages completed!!!~n",[TotalMsg]),
            true;
        true ->
            CatIndex = random:uniform(length(CatList)),
            CatElement = lists:nth(CatIndex, CatList),

            %creation of message
            if
                ID rem 20 == 0 ->
                    %Ran = random:uniform(100),
                    Message = {ID,ID,CatElement,0,0.95};
                true ->
                    Message = {ID,ID,CatElement,0,0.6}
            end,

            A = lists:keyfind(CatElement,1,Popularity),
            {_,PopList} = A,

            PercentageNum = (PerToSeed * length(PeerList))/100,
            IntialMsgPeerList = get_message_peer_list(PeerList,PopList,
                                                       PercentageNum,[]),

            I = ID rem length(Machines),
            Node = lists:nth(I+1, Machines),


            MsgName = list_to_atom("msg_" ++ integer_to_list(ID)),
            message:create(Node, MsgName, Message),

            seed_message(IntialMsgPeerList,Message),
            loop(ID+1, TotalMsg, PerToSeed, CatList, PeerList, Popularity,Machines)
    end.

%% to get list of peers to which messages should be seeded
get_message_peer_list(PeerList,PopList,Percentage,Acc) ->
    if
        length(Acc) >= Percentage ->
            Acc;
        true ->
            if
                (length(Acc) < (0.8*Percentage)) and (length(PopList) > 0) ->
                    Index = random:uniform(length(PopList)),
                    Element = lists:nth(Index, PopList),
                    case lists:member(Element, Acc) of
                        false ->
                            NewList = lists:delete(Element,PopList),
                            get_message_peer_list(PeerList,NewList,Percentage,
                                                  [Element|Acc]);
                        true ->
                            get_message_peer_list(PeerList,PopList,Percentage,Acc)
                    end;

                true ->
                    Index = random:uniform(length(PeerList)),
                    Element = lists:nth(Index, PeerList),
                    case lists:member(Element, Acc) of
                        false ->
                            NewList = lists:delete(Element,PeerList),
                            get_message_peer_list(NewList,PopList,Percentage,
                                                  [Element|Acc]);
                        true ->
                            get_message_peer_list(PeerList,PopList,Percentage,Acc)
                    end
            end
    end.
```

```
%% sending messages to the seed peers
seed_message([],_) ->
    true;
seed_message(PeerList,Message) ->
    [Element|RestList] = PeerList,
    global:whereis_name(Element) ! {seed,Message},
    seed_message(RestList,Message).

%% to get info from the message process
analyze_msg(MsgName) ->
    global:whereis_name(MsgName) ! {analyze}.

%% to get info from the node process
analyze_node(NodeName) ->
    global:whereis_name(NodeName) ! {analyze}.

%% to continue the experiment with SeedMsgNumber of seed messages
continue(SeedMsgNumber) ->
    global:whereis_name(rerun) ! {run,SeedMsgNumber}.

get_nodelist() ->
   global:whereis_name(storage) ! {nodes, self()},
   receive
       {nodelist, List} ->
           List
   end.

get_num_of_msg() ->
   global:whereis_name(storage) ! {messages, self()},
   receive
       {num_of_msg, Num} ->
           Num
   end.

store(NodeList,NumOfMsg) ->
    receive
        {nodes,Pid} ->
             Pid ! {nodelist, NodeList},
             store(NodeList,NumOfMsg);
        {messages,Pid} ->
            Pid ! {num_of_msg, NumOfMsg},
            store(NodeList,NumOfMsg);
        {change_num_of_msg,Num} ->
            store(NodeList,Num)
    end.
```