INF-3990
Master thesis in
Computer Science

Designing and evaluating the SCARF (Scalable CAching layered Recommendation and Feedback powered) system for multimedia content providers

by

Torkil Grindstein

November 2009

Faculty of Science
Department of Computer Science
University of Tromsø

Abstract

This Master thesis covers the design and evaluation of a multimedia content provider service. The design is scalable, both by means of storage, bandwidth and user count. To make the system stand out compared to competitors, there has been added design elements like a recommendation engine and a user feedback system.

The system has been named **SCARF**, which is an abbreviation for **S**calable **CA**ching layered **R**ecommendation and **F**eedback powered system.

In order to do the evaluation part of the thesis, we have implemented a simulator for the entire system, on which we have executed numerous test runs.

Acknowledgments

First of all, I want to express my sincere gratitude to my wife and children for their tremendous patience with me during my work on this thesis, mostly on evenings and nights for a long time. I would never have finished, if it was not for their support.

Secondly, I am honored to thank my supervisor Åge Kvalnes for his most valuable contribution. His insight is tremendous.

Finally, help and support from other project fellows Professor Dag Johansen, Steffen Viken Valvåg, Tjalve Aarflot, Lars Brenna, Tord Heimdal and Robin Pedersen have been outmost valuable.

Thank you all!

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The scenario is a multimedia content provider and a set of users who stream videos in accordance to their respective fields of interests. The content provider's two main interests are 1) to keep users consuming as much content as possible, and 2) to attract as many users as possible. To achieve that, it is beneficial for the provider to be able to offer recommendations to the users, based on their respective interests and behavior history. A lot of research is done within the field of recommendations (e. g., Andersen *et al.* [1], Zhang [46], Shepitsen *et al.* [37]), and in order to provide useful recommendations, the server needs to *know* the user somehow. The two major ways to gain knowledge about users are by *implicit* and *explicit* user feedback (Oard *et al.* [31] and Zigoris *et al.* [48]). Implicit user feedback approaches typically involve monitoring, storing and analyzing user behavior, while explicit user feedback typically involves letting the user explicitly educate the server on preferences and likes/dislikes. These approaches are of course not mutually exclusive, and could (and should) be combined to gain the best possible recommendation engine. Additionally, experiences from *similar users*' behavior are valuable in designing a good recommendation scheme. To define *similar users* is a very challenging task, and examples of relevant research can be found in Nisgav *et al.* [30] and in Gharemani *et al.* [13].

Another important issue the multimedia content provider needs to handle is *load*. Streaming media resources requires a lot of bandwidth, and serving a large number of users places huge demands on the connecting network. A commonly applied approach is to replicate the multimedia content on multiple independent servers that share as few links as possible on the network path leading to the users that are consuming the content.

This thesis addresses these issues, and presents a design with solutions for both the bandwidth limitation problem and the recommendation problem. The bandwidth problem is solved by replication in a caching layer of servers, while the recommendation problem is met with the design of a recommendation engine that combines different recommendation techniques. Among these techniques is the inclusion of a system for collecting feedback from the end users, and applying this feedback in the ranking algorithm.

1.2 Problem definition

The problem area addressed in this thesis is how a multimedia content provider can offer a scalable service, beyond limitations set by own software, hardware and bandwidth. More specific, the required *bandwidth* is too high for a single server, and the provider is unable to afford building up an own server park. We divide the problem area into four different sub-problems:

1. Investigate a distributed caching substrate for efficient use of the network between the content provider and the users.

2. Investigate use of recommendations as a means of input on how content should be replicated across the caching substrate.

3. Investigate a recommendation scheme that balances traditional ranking, user feedback, up-and-coming content, and brand new content.

4. Investigate how the feedback from the end users can be applied to minimize the time it takes to maximize availability for popular content.

1.3 Methodology

Denning *et al.* [9] divided the methodology of computer science into three disciplines:

Theory — based on mathematical principles.

Abstraction — based on experimental scientific methods.

Design — based on engineering.

However, Denning *et al.* argue that the three disciplines of computer science are so interwined that they actually in practice are hard to separate from each other. Trying to categorize the work done in this thesis ends up with a conglomerate of the three disciplines. Theory is used to define the problem, abstraction to construct a solution model, and design is represented in the implementation of a simulation.

1.4 Project context

This thesis was written under an umbrella owned by the *Wide Area Information Filtering* [12] (WAIF) project, which is a joint project between the University of Tromsø [32], Cornell University [43] and UC San Diego [10].

WAIF's slogan is to *get rid of the traditional computer*, and focuses on deriving new architectures and paradigms based on publish/subscribe models instead of traditional client/server session-based interactions, which are most common on the Internet today. An imperative goal is to enhance user experience by implementing intelligent systems based on recommendation engines,

personalized filters and user feedback. Environments of thousands, or even millions of computers, can be available for single user tasks.

The work and results of this thesis is entirely in the WAIF project's spirit, however developed independent of the rest of the WAIF subprojects.

1.5 Outline

The chapters of this thesis are outlined as follows: Chapter 2 discusses related work, with focus on peer-to-peer technologies and recommendation engine solutions. Chapter 3 details the design of SCARF, while chapter 4 focuses on the actual implementation of the complete SCARF simulator. In chapter 5, the testing of the simulation is discussed, while chapter 6 concludes the thesis and gives some pointers to potential future activities, based on conclusions drawn from the simulation results.

Chapter 2

Related work

Traditionally, architectures for file and information sharing have been centered around the client/server model. This concept arose in the 1980s [21] and describes the different roles PCs had in small scaled computer networks. The client is a requester of services, and the server is the provider of services. This is still the most applied model for business applications being written today, as well as for the main application protocols used on the Internet, e.g., HTTP, SMTP and FTP. Details on the client/server model are presented in Schussel [36] and Edelstein [11].

2.1 Peer-to-peer

The client/server representation is a very intuitive model of how to retrieve wanted data or information on a network. However, given situations where very many clients access the same server, or situations where retrieval of data requires large amounts of bandwidth, then both the server and the server's network become bottlenecks. Consequently, the clients do observe poor quality of service.

This is the background for the peer-to-peer (P2P) paradigm. The idea is as follows: If one client has downloaded data from a server, then it might take the role as a server for other clients demanding the same piece of data. This reduces the load on the main server, as well as distributes bandwidth requirements throughout the network. As a side note, the P2P model was the original system for the infant Internet back in the 1960s. In 1969, ARPANET, which is considered the aboriginal Internet, was designed as a network of equal peers, rather than a network of servers and clients [17].

In P2P systems, a major task to solve is to find a way for end users to actually find the desired content. A *distributed hash table* (DHT) is a decentralized lookup service, which distributes the responsibility for maintaining the service among nodes in the system (Balakrishnan *et al.* [3], Kaashoek *et al.* [23] and Monnerat *et al.* [28]). The first four DHTs were introduced more or less concurrently in 2001: Pastry (Rowstron *et al.* [35]), Tapestry (Zhao *et al.* [47]), Chord (Stoica *et al.* [40]), and the Content Addressable Network (CAN, Ratnasamy *et al.* [33]).

The DHTs arose as a reaction to experiences with weaknesses in large P2P systems, like the music sharing systems Napster (Carlsson *et al.* [7]) and Gnutella (Ripeanu *et al.* [34]), as well as the anonymity provider Freenet (Clarke *et al.* [8]).

This is how these three systems addressed data lookup:

Napster used a centralized index server. When new peers joined the system, they provided the central server with a list of locally hosted files. This list was then merged into the central index. A central component is a single point of failure, and this design also made the system vulnerable to attacks and lawsuits.

Gnutella avoided the single point of failure problem by basically broadcasting each search query to all other nodes in the network. However, this approach was significantly less efficient than the Napster approach.

Freenet also distributed the lookup process, and employed a heuristic key based routing model. Here, each file was associated with a key, and files with similar keys tended to cluster. It turned out that content usually could be found in few steps, but the protocol could not guarantee that wanted data would be found, even if present in the network.

2.2 Recommendations

Recommender systems typically employ *information filtering* techniques to suggest information items (books, movies, music, Internet sites, etc) to users, based on the system's knowledge of user preferences.

This knowledge can be obtained either by explicitly *telling* the system about the user's preferences, e.g., by filling out forms, by letting the user rate known content, or by implicitly *learning* user preferences, e.g. by monitoring the user's activity and behavior, or by employing knowledge gained by monitoring user behavior from users that are supposed to share interests with the actual user, i.e. *similar* users. Combinations of explicit and implicit learning can be applied.

In cases where systems make recommendations based on similar user behavior, there is one underlying basic assumption: **Users with similar preferences in the past are likely to share preferences in the future**.

Back in 1992, Goldberg *et al.* [14] introduced the concept of *collaborative filtering*. It was a method for users to collaborate on a way to evaluate the degree of interest on incoming mail in their *Tapestry* mail system. David Goldberg and his team derived revolutionary algorithms, which later have been refined and a lot of research have been done within the field ever since. With the tremendous growth of e-commerce and social networking, the topic is hotter than ever.

2.2.1 Exempli gratia

On the Internet today, there exist numerous examples of how different recommendation systems are employed – some are subtle, others very obvious.

A newspaper front-page is a simple recommendation example. An editor makes choices on behalf of an average reader's interests, and neither explicit nor

implicit user feedback is necessarily used. The editorial choices are probably made based on a combination of several variables, like freshness, estimated user impact and locality. However, some on-line newspapers (e.g., The New York Times [42]) offer their readers to open personal accounts, and hence make both explicit (where the user states personal interests and/or rates articles) and implicit (logging the articles read by the user) learning feasible. In this way, users with own accounts can be presented with a totally personalized front page for the newspaper.

Another popular service on the Internet is the *Last.fm* [26] music service. Last.fm monitors the music users listen to, and derives recommendations along several axis. Most obvious, Last.fm generates charts based on the most listened to music during some preset time intervals (last week, last month, last year, etc.). Within this category, Last.fm actually create numerous charts: "Top artists" (most listened to artists), "Hyped artists" (artists with most increased popularity recently), "Loved artists" (artists that users explicitly has flagged as "loved"), and also all of the three mentioned chart types applied on specific songs instead of artists.

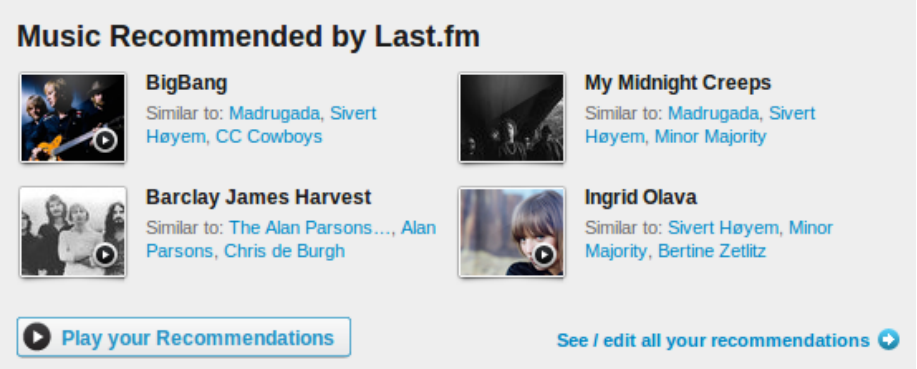

Figure 2.1: *Screen shot from the Last.fm web page. Here are presented specific recommendations for the logged in user, based on an analysis of the monitored user behavior, and a complex music categorization system.*

Furthermore, Last.fm have devised a wide spanning categorization system for music, which simplified makes music taste *measurable*. This can be done by attaching large amounts of *metadata* to each piece of content (artist or song) — some manually and some algorithmically. (See figure 2.1.) Values of these metadata fields can be stored and structured in a vector. This makes it possible to compare different users' music taste, and hence provide music recommendations to users based on similar users' taste. To do this comparison, algorithms for calculating mathematical vector distance can be applied (e.g., *Euclidean* distance [4]). Last.fm automatically generates a *group* of similar users, called the user's *neighborhood*. Additionally, the user can manually compile a group of friends, which makes it possible to compare music taste with friends, and hence discover music previously unheared of.

The Internet service that is most similar to the SCARF system presented in this thesis is *YouTube* [45]. YouTube is a video streaming provider that streams

to millions of users, also creating recommendations based on similar content. YouTube's solution to the scaling problem is to add disks, servers, and even server parks on different locations. Hence, one of YouTube's greatest challenges is to generate software on top of the huge amount of hardware — software that handles millions of users, and still serves user requests in a matter of milliseconds.

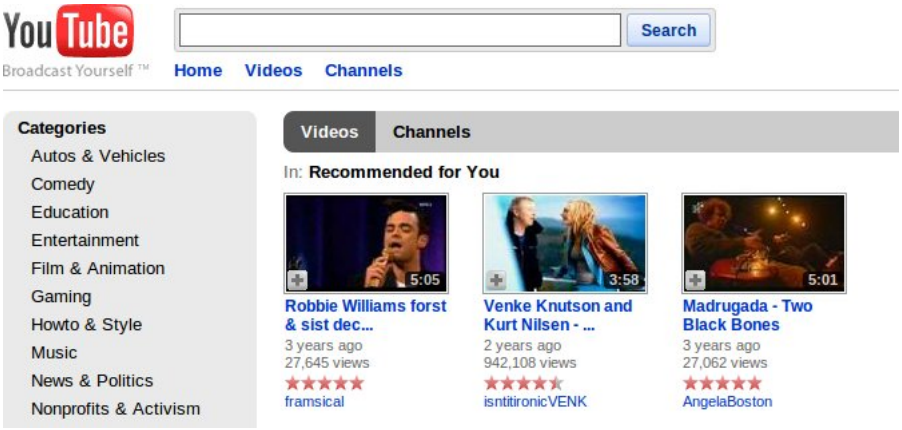

Figure 2.2: *Screen shot from the personal YouTube recommendation page.*

One of the most recent services that has gained massive popularity is *Spotify* [39], which hosts an enormous database of music. Millions of users stream music on demand, and the music might in the process be cached on the client computer. In the next step, this client can be the server for other clients requesting the same song. Hence, we see that Spotify have addressed the bandwidth bottleneck with a peer-to-peer based solution. This caching scheme is processed totally invisible for the users. There are, of course, numerous issues the developers must take care of in such a solution to prohibit unwanted distribution of copyrighted material. However, these issues are beyond the scope of this thesis to discuss.

2.2.2 Categorization of recommender systems

Burke [5] divides recommender systems into five main groups:

1. **Collaborative-based recommendations.** A collaborative-based recommendation system is based upon finding similarities between users. In order to do that, there must be generated a profile for each user. The recommendation itself is then based upon comparing and matching different user profiles. Users in systems like this typically rates items – ratings that will be attributed to their user profile. A weakness with this approach is that new unrated items are not automatically entered in the recommendation process – it needs to be found and rated by a user first. Another weakness is that users with unusual tastes might suffer from a relatively empty dataset with which to match. Among the advantages with this approach is that it improves over time. The more data, the better

recommendation engine. Collaborative-based recommendation systems might also discover cross-genre niches. An example of a relevant system is Phoaks [41].

2. **Content-based recommendations.** Content-based recommendations is based on matching items with similar attributes. In this approach a user profile is also applied, attributed with items highly appreciated by the user. For matching of items to be possible, information must be textually extracted from the items. Hence, this type of recommendation is best suited for textual based documents, and not for binary documents (like music, images and videos). An advantage with content-based recommendation is that it improves over time – the quality of the recommendation improves with the amount of metadata. InfoFinder [25] is a system that employs content-based recommendation.

3. **Demographic-based recommendations.** Based on the assumption that people with a similar demographic profile share interests, it is possible to produce profile specific recommendations. A demography profile can be built on information like age, gender, location, education level, etc. The idea is that people with similar background are likely to share interests. A weakness of this approach is that the assumption is not valid for all users. There are aberrancies within each demographic group. Like the other recommendation techniques discussed above, this approach also has the advantage of improving over time, as the datasets grow. An example of a demographic-based recommendation system example is Lifestyle Finder [24].

4. **Utility-based recommendations.** In a utility-based recommendation system, history is not important. Each recommendable item is attributed with different utilities, and the user's *current* need is matched with these metadata. Overlooking history can serve as both a weakness and an advantage. It is a weakness because user behavior tends to repeat, and it is an advantage in cases where user behavior and interest evolve and change. It is easier to cope with user behavior changes. Several advanced methods to provide this type of recommendations can be applied, and Huang [20] compares some of these.

5. **Knowledge-based recommendations.** In a knowledge-based recommendation system all items are attributed with knowledge on how the item meets user specific needs. This approach does not either learn from history, which might be considered both a weakness and an advantage. A system like this is also dependent on that editors do a lot of work to insert knowledge to the system. Examples of systems employing this recommendation technique are intelligent on-line travel agencies and television recommender programs. Shi [38] has implemented a system of the latter category.

2.2.3 Privacy issues

Using individually adapted recommendations has its obvious advantages as discussed above, but one must be aware of a potential misuse of the personal data the recommendation engine stores on your behalf. The knowledge the recommendation engine acquires when learning to know your preferences and tastes might be distributed to specialized advertisement providers, who use this knowledge to generate personalized advertisements just for you. These aspects of personalization contain ethical regards the users must be aware of. Even though practically nobody really reads the personalization disclaimers of service providers prior to establishing an account, one is strongly encouraged to do so.

Chapter 3

Design

`SCARF`, which is an abbreviation for `Scalable CAching layered Recommendation and Feedback powered`, is a system designed to handle the needs of a multimedia content provider that has too high bandwidth requirements for a single network ingress/egress server installation. Although SCARF has been instantiated in the form of a simulation engine, the simulation makes detailed assumptions of an underlying system design. In this chapter we detail this design.

Overall, SCARF employs a combination of three strategies in its design:

1. **Employ a caching substrate.** The idea is to distribute load from the content provider to a number of nodes that can serve as the user entry point to the server. Users will consume content from one of these nodes instead of directly from a single server at the content provider.

2. **Employ replication.** We assume that, due to varying popularity, a single node in the caching substrate is unable to serve all potential consumers of a piece of multimedia content. Thus, content must be replicated to increase the potential bandwidth for serving popular content. As a consequence, an algorithm is needed to guide how the content is replicated across nodes in the caching substrate.

3. **Employ recommendations.** There exist several multimedia providers on the Internet. In order to attract users to this particular service, we need to let the users find what they want as soon as possible, ideally sooner than the competitor services. Therefore, SCARF integrates recommendations as part of its basic architecture – recommendations are used both to guide replication and as a means to guide users to content that may be of interest.

3.1 Architecture overview

The architecture we have devised contains three main components, as can be seen in figure 3.1.

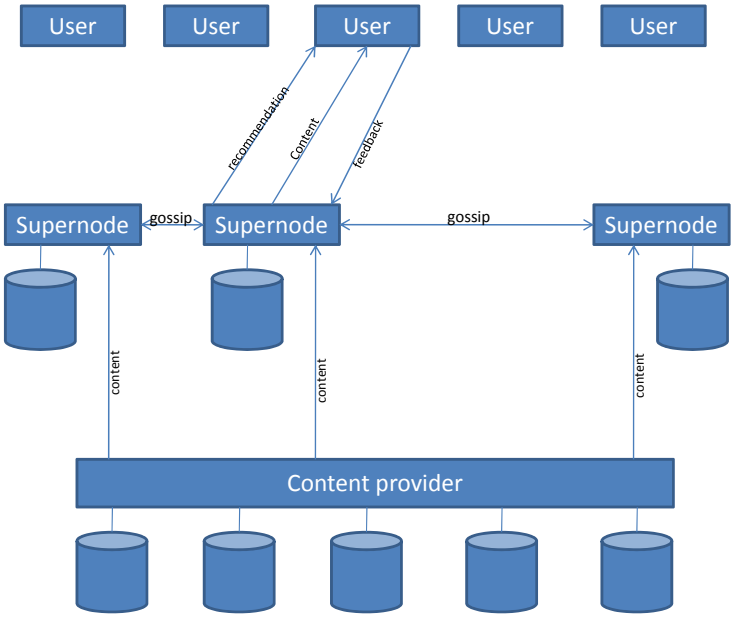

Figure 3.1: *A simple overview of the components in the system.*

First, we have the **content provider**, which is responsible for storing the entire corpus of multimedia content. New content initially enters content provider storage. Second, we introduce a caching substrate consisting of **supernodes** that serve content to consumers (users). To make a piece of content available to users, the provider pushes the content to a supernode. The provider may push content to more than one supernode initially, depending on assumptions of immediate popularity.

Supernodes are organized in a peer-to-peer structure. Each supernode has the following characteristics:

1. Ability to store a subset of the entire multimedia store from the content provider.

2. Ability to connect to a set of users interested in multimedia content. A supernode should not be connected to more users than it can handle, mainly restricted by bandwidth limitations.

3. Ability to provide recommendations to users. The recommendation is based on the search query specified by the user, and works along three different axis:

   (a) **Rating** – a combination of traditional rank provided by the content provider, and feedback from users who already viewed this content,

   (b) **Up-and-coming content** – content that has received *recent* good user feedback, but not necessarily reached the rating level needed to compete with content in the rating axis, and

   (c) **Fresh content** – content that is very recently added to the local store, not necessarily viewed by anyone yet.

   This recommendation scheme is thoroughly discussed in section 4.2.10.

4. Ability to spread gossip to fellow supernodes. This means that when the supernode detects that it serves content that is on a particular popularity level, it notifies all the *neighbor* supernodes about it. The supernode then allows these neighbors to download the content to their own local stores, if not already present. The gossip scheme is elaborated in section 4.2.9 and evaluation of different neighbor architectures is found in section 4.2.2.

5. Keeps an own search index of its stored content. This is dynamically updated, based on input from users, other supernodes and the content provider.

Finally, we have the **users**, who basically consume multimedia content streamed from a supernode. SCARF does not incorporate a scheme for how a user locates a particular supernode. In our simulations this is done randomly. Commonly applied approaches to this problem include DNS rotation [2], a content provider web page that dynamically routes and assigns a user to a server (in our case, a supernode), and others electing clients as supernodes [27]. After watching a video, the user determines whether this was *good* or *bad*, and

sends this feedback to the providing supernode. Upon selecting next video to watch, the user selects at random a video from one of the three recommendation axis.

As stated in the introduction chapter, our approach in this thesis has been simulation. To exemplify and clarify, a simplified mock-up example on what the users' interface could look like is shown in figure 3.2.

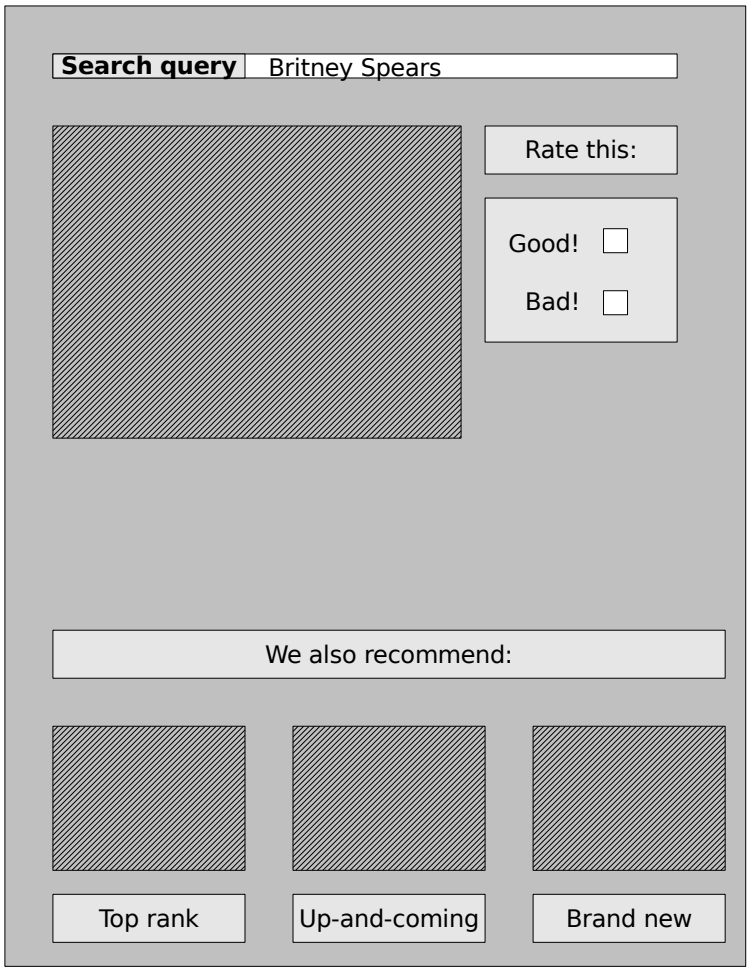

Figure 3.2: *A mock-up example on what the users' interface could look like. On the top, there is a field to specify a search query (also referred to as* topic*). Next comes the currently streamed video, with the feedback functionality to the right. On the bottom, the three different recommendations are shown.*

3.2 The supernodes

The three strategies discussed above chiefly involve the supernodes. In this section we detail aspects of the supernode design, their capabilities, and their

responsibilities.

3.2.1 Content administration

Each supernode hosts a subset of the entire multimedia storage. There are two ways a supernode can receive new content to its storage:

1. When the content provider updates its main repository of content with new content, it pushes this to a subset of its supernodes.

2. When a supernode receives gossip about content from another supernode, it may decide to download this content to its own storage.

When the supernode's storage is approaching its limits, it will start discarding videos, and need to be sure that the discarded videos are ranked low along all three axes.

3.2.2 Query interface and content indexing

On the supernode, all content can be looked up in a structured **search index**. The search index lets the supernode look up all content based on particular search terms.

The supernode annotates content with value attributes that can be used to provide recommendations to the end users. Furthermore, the recommendation engine has three different *views* of the search index – each sorting the content according to the different recommendation axes.

By introducing a query interface, and hence allowing the user to specify search terms, the user will find the wanted content faster than general content browsing/directory lookup. The SCARF design itself does not exclude directory based browsing. That could simply be implemented by queryless searches hidden within each click.

3.2.3 Communication

The supernodes communicate with several parties in the architecture. These communication peers are either the content provider, other supernodes or end users.

The communication with the content provider mainly involves receiving content pushed from the provider. Another type of communication, which we have not focused on in the simulation, is to propagate aggregated user feedback or view statistics back to the provider. From the perspective of the content provider, obtaining such information can be useful, for example to adjust the global ranking for all content (see section 4.2.4), or to integrate into a pay-per-view scheme.

The communication with other supernodes involve gossip exchange and potential downloading of new content. Gossip messages are sent whenever a supernode detects that a particular video seems to be of high interest among its users. Upon receipt of gossip messages, a supernode checks whether the

particular video resides in its local store. If not, it will download the video from either the referring supernode or from the content provider.

The communication with end users involve:

1. **Search** The user issues a search query to the supernode.
2. **Recommendation** Based on the search query and the local search index, the supernode returns three recommended videos – one highly ranked, one with recent high popularity, and the most recently added video (relevant for the search query). These types of recommendations are throughout this thesis referred to as the three *axis* of recommendation.
3. **Streaming** Upon receipt of the video recommendations, the user selects one, and starts watching. The watching is done by streaming the content from the supernode.
4. **Feedback** After watching the video, the user rates it, and the rating is sent back to the supernode.

When the supernode receives the feedback from the user, its search index is updated accordingly, and if a *popularity trigger* is fired, gossip about the content is spread to other supernodes. How to define the popularity trigger is not trivial. First, some minimum number of users must have given feedback, and the positive feedback percentage must be above a certain level. To optimize these parameters needs a lot of research on real world data, and the optimal values are perhaps impossible to derive. In the test simulations presented in chapter 5, we have defined the trigger to fire if at least five of the seven first viewers gave positive feedback. It is easy to change these to any values the simulator administrator wants to try.

3.2.4 Neighbor topologies

We have argued about the importance of having a caching layer between end users and the content provider. But we have not discussed how this substrate should be internally organized. Intuitively, one may think that a fully connected network of supernodes is the best, as the distance between any supernode is just 1 hop. Then popular content will have a potential to reach fully cumulative bandwidth very quickly. However, having this topology also floods semi-popular content in the same speed. It just needs to fire *one* popularity trigger to reach the entire network. This might create a lot of unnecessary traffic on the network, and the supernodes will need to process discard routines more often. We believe that running a series of experiments might provide some knowledge on which topology that yields the best trade-off between distribution rates and traffic. We did such experiments with the SCARF simulator, and the results can be observed in chapter 5.

The SCARF simulation engine accommodates four different topologies for how the supernodes in the caching substrate are organized. These include:

1. **Ring** The simplest topology is the ring topology. In this case, each supernode is connected to exactly one other supernode. So, if a trigger

in supernode $\mathbf{S}_n$ regarding content **C** for topic **T** fires, it notifies only supernode $\mathbf{S}_{n+1}$ about this.

2. **Double ring** In this topology each supernode has two neighbors. Supernode $\mathbf{S}_n$ will be connected to $\mathbf{S}_{n-1}$ and $\mathbf{S}_{n+1}$. This topology is sometimes referred to as a *bidirectional* ring.

3. **Two rings** In a two ring topology, each supernode has two neighbors, but as parts of two different ring definitions.

4. **Fully connected network** In the fully connected network topology, each supernode is connected to all the others.

There are several more topologies that could be investigated in future versions of the SCARF system.

Chapter 4

Implementation

Setting up a complete running installation with thousands of machines and users is outside the scope of this thesis. To verify algorithms and analyze effects of the SCARF design, we have therefore implemented a simulation of the entire system. This simulation handles tens of thousands of users, simultaneously streaming content of different topics from hundreds of supernodes, with a layer of supernodes providing users with recommendations and other supernodes with gossip. In this chapter we detail the implementation of the SCARF simulator.

4.1 Platform

The simulator is written in the java programming language, v. 1.6. All output from the simulation are distributed in several formatted text based log files, to ease monitoring and execution evaluation. The simulator code can be found in appendix A.1.

4.1.1 Simulating time

The SCARF simulator employs the notion of *simulated time*. As long as we apply simulated time, the time units need not be close to real time measurements at all, as long as events occur in correct order *relative* to each other. These issues were introduced by Hoare [19] back in the seventies, and they were thoroughly discussed in the Winter Simulation Conference in 1986 [18].

Each unit of time encompasses actions, such as:

- a user starting to watch a video,
- a user finishing watching a video,
- a user rating a video,
- a supernode determines to spread gossip to other supernodes,
- a supernode receives gossip from another supernode,
- a supernode is updated with a new video, and

- new content arrives to the system,

For performance, two different time simulation schemes are implemented in SCARF. The first has a virtual clock which ticks once per iteration, and runs through a queue of ongoing events, counting them down until termination. This approach has the advantage of making it easy to manually monitor system behavior, and one can implement more user friendly monitoring applications on top of it. However, this method scales very badly, since it requires a lot of processing. For instance, watching a video of length 200 will require at least 200 slots in the simulation.

The other approach to time simulation is commonly known as a *Discrete event simulation* [18]. It works as follows: Each ongoing event is queued in an event queue, sorted by expected time of termination. The *event handler* picks an event from the beginning of the queue, and executes it in accordance with the rules for that particular event type. The event processing might imply insertion of one or more new events in the event queue, and these events find their place in the event queue based on their expected time to occur. This method is less computationally intensive than the first, as the number of calculations is significantly decreased, and the above mentioned example with a video of length 200 will now only require 2 slots in the simulation. All tests described in chapter 5 use a simulator configuration of this type.

4.1.2 Simulating popularity

Writing simulator code for protocols and server algorithms is not the most challenging task when writing a simulation for a large system. One of the hard things to simulate is human behavior. In the SCARF system, this includes the rules for when a user will rank a video as good or bad.

Guo *et al.* [16] claim that it is commonly agreed that web traffic follows the Zipf-like [29] distribution. Figure 4.1 shows the zipf distribution, both by linear and logarithmic scales. However, media file access does not necessarily follow the patterns from general web traffic. Guo *et al.* point to numerous empirical research projects whose results are not conform, mostly because the analyses are done on too small data volumes. Their own research is based on eight years of data collection, and covers up to hundreds of millions of client requests. Their conclusion is that multimedia web traffic tends to follow a *stretched exponential distribution* graph.

However, user behavior has evolved during this eight-year time period, and saying that popularity is a function of access quantity is not valid. This is because availability and recommendations also highly influence what users watch. A video with a the highest popularity potential might have never been seen by anyone, just because it can not be found.

Furthermore, assuming that absolute popularity is possible to predict, it is still next to impossible to deduce a generic formula for how to set this value in a dynamic system. One could deduce a probabilistic function that generates popularity values along the graph of such a function, but what will that value really express? In SCARF, the users are given one boolean way of providing

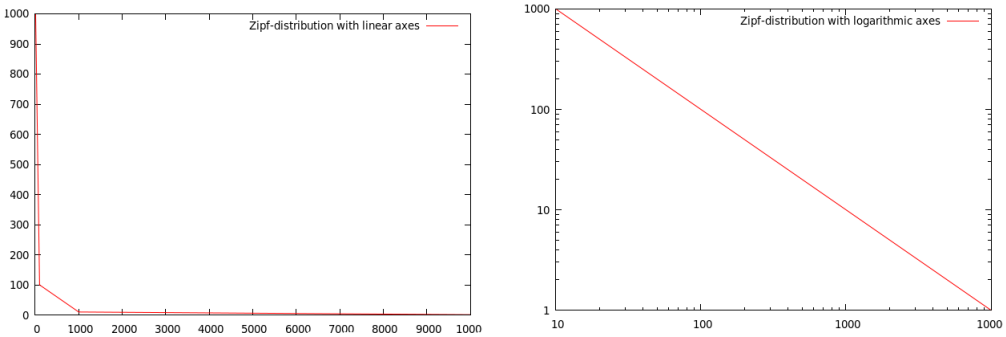

Figure 4.1: *A zipf distribution graphed by linear scales (left) and logarithmic scales (right).*

feedback – either *good* or *bad*, and we want *all* users to be able to reach as much popular content as they want. We do not want the *next most popular* video to be available for a reduced set of users.

The conclusion of this discussion is that it is not most important to generate a popularity function as realistic as possible. The important thing is to just *have* a popularity value, and analyze what this value does with the execution of the entire SCARF system. We have implemented a hidden attribute for each multimedia content, which serves as the popularity value, and this value is simply a random number between 0 and 1. When a user have viewed a video, it is generated another random number between 0 and 1, and if the former is higher than the latter, the user will rate the video as *good*. It is simple, but it still works as a reference for popularity. Nevertheless, the important contribution from SCARF is how to *act* upon the user feedback.

4.2 Architecture

The SCARF design was described in chapter 3. Here we describe in more detail how the SCARF simulator implements all aspects of the design. The following subsections will describe the system's main actors in more depth.

4.2.1 The content provider

As seen from figure 3.1 the content provider might look like an important entity of the architecture. However, for simulation purposes, the role of the provider can be reduced. The simulator assumes that the provider just stores a large number videos, as well as once in a while to update its store with new content. Hence the content provider is implemented more like a *stub* to the supernodes in the system.

4.2.2 The supernodes

A supernode is defined by *The Free Dictionary* [6] as *A user's computer in a peer-to-peer network that acts as a relay between one user and another. Super-*

nodes are typically established automatically by the peer-to-peer software based on current network traffic and the capabilities of the user's machine. Users may have little or no control whether their machines become supernodes, or they may have the option to not participate. In SCARF, the supernodes are defined slightly different. Here they constitute a one dimensional layer between the content provider and the end users. How they are chosen to serve as supernodes have not been thought of as relevant in our simulator. Lo *et al.* [27] discuss such election protocols in depth.

Each supernode has the following characteristics:

id A unique identifier throughout the system. Converted to a real world implementation, this would represent a unique URI, but in our simulated environment these are just positive integers.

users Each supernode is associated a predefined group of end users. Consequently, each supernode needs a way to lookup, and address each end user. This is done by keeping an array list of all connected end users in the supernode class.

content All supernodes contain an evanescent store of multimedia content. This must be stored in a way that it easily can be addressed upon request, both from end users and from other supernodes. The content must be stored in a globally coherent namespace, to avoid misinterpretations between supernodes, end users, and the provider storage.

index Each supernode administrates its own search index. They play by the same globally set of rules, but as they *know* a different set of users and contain a different set of contents, these indexes evolve independently of the others. The index management is more thoroughly discussed in section 4.2.5

neighbors Supernodes exchange information based on what they learn from their attached end users. Each supernode must know where to find the neighboring supernodes. How many neighboring supernodes each supernode has depends of the topology under which the simulation is run. If the topology is set to be a ring network, each supernode has one neighbor (if the ring is directed) or two neighbors (if the ring is two-way). In a fully connected topology, each supernode is connected to all the others. Results of all testing show that the time it takes for popular content to spread to all supernodes decreases with an inverse proportional ratio to the number of connected neighbors.

communication Finally, the supernodes need communication interfaces both to the end users and to the neighbors. That is, the end users must know how to specify search terms, and how to provide feedback on viewed content. Additionally, the supernode needs an implementation on how to present the recommendations to the end users. The supernodes must also have some way to convey own experiences of given content feedback to the other supernodes – the latter known as *gossip*.

In the implementation there are other parameters as well, like log file location, index file location, boolean parameters, etc, but these are so implementation specific that they are negligible for this discussion.

Neighbor topologies

There are numerous of ways to order the supernodes, and the SCARF simulator implements four different topologies.

1. `RING` — The `RING` topology orders all supernodes as a chain, each connected with the next supernode, but not vice versa.

2. `DOUBLERING` — In the `DOUBLERING` topology, all supernodes are ordered in a chain, with connection both ways.

3. `TWORINGS` — This topology implements two different (one-directional) rings, where each supernode has its place in both rings.

4. `ALLCONNECTED` — The topology where all supernodes can communicate directly with all the others.

In the tests described in chapter 5, all topologies are tested.

4.2.3 The end users

The simulation of the end users is the only part of the entire system that tries to simulate actions of human beings. In order to do that in a satisfactory manner, we need to make some assumptions, and define some limitations. These assumptions and limitations must not be defined in a way that compromises the model.

The assumptions and limitations can be seen by studying these most important characteristics of the end users:

id All users must be uniquely defined. For simplicity, they are defined by positive integers.

connection Each end user addresses one and only one supernode throughout the simulation. As discussed in section 6.2 it would be interesting to do some future investigations on performance improvements with a more dynamical relationship between supernodes and end users, but SCARF defines this as a fixed relationship. Hence, a user must know where to find its supernode.

interest Every user is supposed to issue a search query to its supernode. This search query is based on the user's current topic of interest. In order to be able to control the number of parameters and in order to have enough focus to compile valuable simulation results, we have defined a limited vocabulary from which the search queries can be fetched. Test rounds have been run with a vocabulary of just one word, and up to twenty words. Results show that the size of the vocabulary has no particular

impact on the simulation result, and hence we limit all end user to do the very same query throughout the entire run[1].

content The end user will, after a few simulated seconds of decision making, choose which content to watch, and our end user representation must constantly know a) what piece is currently being watched, and b) which content has been viewed earlier. How recommendation is done and how the user chooses between the recommendation axes is addressed in section 4.2.10.

4.2.4 Content

When the concept of *content* is used in this thesis, we always mean a multimedia video available from the content provider, and also available from a subset of the supernodes.

Most important characteristics are:

id Again, everything needs to be uniquely identified. We keep it simple, and use running numbers.

rank Since SCARF can be placed on top of an existing static ranking scheme, we assume that every video already has a rank based on traditional ranking algorithms. This initial rank is used as an important input to one of the three recommendation axis — the *rank recommendation* axis. When content is uploaded from the content provider to a supernode, the initial ranking is attributed to the content. When that is said, as in the real world, the computized rank does not necessarily reflect the actual popularity of the content. Hence, user input must also serve as input to the the rank recommendation. And as users rate videos, an attribute known as *smart rank* is constantly updated. See section 4.2.10 below for a detailed description of smart rank.

topics Every piece of content is tagged with search terms for which the video is thought of as relevant for the end user. These search terms constitute a subset of the topic vocabulary discussed in section 4.2.3.

length Each video has a length parameter which the time simulation system needs to know. In the implementation, each piece of content has a length in the range from 10 to 300 time units – which for easier understanding might be thought of as seconds. It is assumed that the end user watches the movie to the end, prior to ranking it.

popularity In order to not make the user ranking completely random, and hence practically useless, we have devised a hidden parameter to represent the real popularity among end users. This parameter is an important input to the algorithm for letting the end user rank viewed content. Popularity simulation was discussed in section 4.1.2 above.

[1] That is, one particular user does not alter his/her search query, but two different users might issue two different search queries.

distribution The same content will be distributed among the supernodes, and one of the important missions with the entire simulation is to make sure that popular content is distributed broader than less popular content. This distribution characteristic is also a way of measuring a content's cumulative bandwidth.

4.2.5 The indexes

In SCARF, all supernodes administer their own search indexes. This is one of the most advanced data structures in the entire system. The search indexes are sorted on *topic*, which in practice is the same as a *search query* specified by a user. For each topic, the supernode carries a list of *index elements*, which in turn consist of:

content id The unique identifier for a piece of content. This identifier is globally unique[2].

base rank Each video is *a priori* ranked based on traditional ranking algorithms. This traditional ranking value is stored in the index as the *base rank*. When we start up our simulator, this is the only known ranking we have on all content. For more details about the start-up and initialization, see section 4.2.12. We do not consider calculation of this initial rank within the scope of this thesis, so we simply employ a random based algorithm. The base rank value is normalized to an integer between 0 and 99.

smart rank Since the SCARF system monitors user feedback on every video shown, we have the opportunity to use this feedback to calculate an improved ranking value. We use the base rank discussed right above as the start-up value for the *smart rank*. When a user ranks a video as *good*, we increase the smart rank value, and when a user ranks a video as *bad*, we decrease it. Doing it this way, it is easy to monitor how really popular pieces of content migrate higher on the smart rank than less popular content. (Remember, each content has a hidden attribute containing a value for the real popularity among end users (see section 4.2.4).) The *amount* of increment and decrement is, however, not a trivial task to determine. We have increased and decreased the smart rank value by 10 per feedback, but setting an optimal value[3] for this might be set as a task for future research. Probably, this number is impossible to find in a simulated environment, and relies on subjective opinions.

counters We count each positive feedback and each negative feedback for all videos. These counters are later used by the supernode to determine whether to tip other supernodes about it. This *tipping* is what we refer to as *gossip* in SCARF. For an elaboration, see section 4.2.9.

[2]By *globally* unique, we mean that when two (or more) supernodes contain content with the same id, we speak of the very same piece of content.

[3]The incrementor/decrementor value need not be a constant, but can of course be calculated as a function of the base rank, or of the previous value of the smart rank.

timestamp Whenever a new piece of content is added in the index, the supernode attaches a timestamp for this event. Beware that this is not the timestamp for when the content was added in the content provider's main server, nor is it the timestamp for when it was added in the first supernode. It is just the timestamp for when the video was added to local storage (and hence totally new for all end users belonging to the supernode). This timestamp is used whenever the supernode provides recommendations along the freshness axis.

4.2.6 The topics

The topic class do not contain any other information than a unique identifier and a topic name, also known as search query. When the system is booted, a predefined list of topics is submitted. This does not reduce the power of the simulation, but enhances the ability to monitor the system's behavior. Every supernode stores its search index based on topic.

4.2.7 Time

The time simulation is an essential driver of the system, and is discussed in section 4.1.1.

4.2.8 User feedback

For simplicity, the user has the choice of rating content either as *good* or *bad*. This feedback is as earlier explained used to calculate the *smart rank* for the content. In a more advanced version, we could of course define a broader scale on which to give feedback, e. g., dice throws (1 – 6), or a scale from 1 – 10. This might improve the smart rank, but not necessarily.

4.2.9 Gossip

In a computer communication protocol context, the term *gossip* is normally defined as information exchange with randomly chosen peers (e. g., *Astrolabe* [44] and *T-Man* [22]), but in SCARF the term is used slightly different. In SCARF, the receiver of gossip messages are predefined, as the system topology is predefined. That is, all simulations are run on top of different predefined topologies (e. g., *ring*, or *fully connected* networks). The term *gossip* is chosen because of its semantics. Whenever all user feedback on a particular piece of content **C** meets certain criteria, a supernode **S** notifies each of its *neighbors*[4] $\mathbf{S}'_1, \ldots, \mathbf{S}'_i$ about that **S**'s experience with feedback from users interesting in topic **T** is that content **C** is very good.

From the other side, whenever $\mathbf{S}'_1, \ldots, \mathbf{S}'_i$ receive this message from **S**, they immediately check their local indexes to see if they have the recommended video already stored. If not, they simply request to download the video from the content provider (or the referring supernode), and add it to the local index. They

[4] as defined by the current topology.

do not know that their users have the same preferences as the users belonging to **S**, so the new content will be added with *smart rank* identical to the *base rank*, feedback counters set to zero, and timestamp set to local download time. Consequently, the video will be top ranked on the freshness axis. Given similar feedback from their own users, this will also trigger the gossip criteria to be met, and the epidemic is on its way.

So, to clarify, a supernode can recommend content to its neighbor supernodes, which is referred to as gossip. When the supernode recommends content to its end users, we refer to this as recommendations. The latter is discussed in section 4.2.10 below.

4.2.10 Recommendations

The three axis of recommendation have been referred and alluded to earlier. Every time a user makes a search in a supernode's index, the supernode makes three ranking lists of all relevant content:

1. **Smart rank**. As explained in section 4.2.5, the supernodes constantly calculate a smart rank for each content piece (based on topic). So, when determining what piece of content to recommend along this axis, it just picks the one with the highest smart rank. Note also that the supernode knows which videos each user already has viewed[5], so these are excluded from the ranking.

2. **Up-and-coming**. Based on the user feedback during the last period of time (e. g., the last 10000 time ticks), the supernode is able to rank all relevant content on what has been highly rated recently. This will allow content that erroneously was given a low base rank to still gain attention among the users.

3. **Newcomers** The third recommendation axis is just a ranking of when content was added to the index – fresh content. If the content turns out to be bad, it will soon be ranked down on the two other axes, simultaneously as it of course loses freshness. If the content turns out to be really appreciated by its watchers, it will gain credit in the two other axes while losing freshness points, and consequently still get attention from future viewers.

For simplicity, the users are programmed to randomly choose their videos from the three recommendation axes. Real human users will probably prefer one of the axes to the others, but eventual skews is assumed to be negligible.

To classify the three recommendation axes in accordance with the definitions of the five different classes of recommender systems discussed in section 2.2.2, we will say that the smart rank recommendation axis and the up-and-coming recommendation axis generally belong to the content-based recommendation class, while the newcomers recommendation axis implements a utility-based recommendation class.

[5] See section 4.2.3.

4.2.11 Events

The simulator is driven forward by a series of events. These are queued in a chronological order. Each event are tagged with an *event type*, and all events are among an assortment of 10 types.

The 10 event types are as follows:

1. `USER_STARTS_WATCHING`. When a user wants to watch a video, this event is triggered. There are two possible outcomes of this event type:

   (a) The supernode has provided at least one recommendation in accordance and response to the search query. Then the user starts watching this video, and a new event of type `USER_STOPS_WATCHING` is created and put on the event queue.

   (b) If the supernode is unable to recommend any video in accordance with the search query, then a new event of the same type (`USER_STARTS_WATCHING`) is created and inserted in the event queue. This simulates the user waiting for a random amount of time, and then tries to search again, hoping that the supernode has received new content within the field of interest.

2. `USER_STOPS_WATCHING`. This event is triggered when a user is finished watching a video. When this event is processed, the user will give a feedback on the video – either *good* or *bad*. If the feedback is good, the supernode must check if the criteria for sending gossip to neighboring supernodes are met. If the criteria are met, a new event of type `SUPERNODE_SPREADS_GOSSIP` is queued. The simulator can also be run in a verbose mode, and then there will be triggered a `SUPERNODE_RECALCULATION` event in this step. The last event generated as a consequence of `USER_STOPS_WATCHING` is a `USER_IDLE` event, which just simulates the time period the user needs to decide which video to watch next.

3. `USER_IDLE`. This is an event type used to simulate time periods when the user is considering alternatives. Outcomes of this event are another event of the same type, or an event of type `USER_STARTS_WATCHING`.

4. `SUPERNODE_SPREADS_GOSSIP`. When a supernode discovers that the criteria for gossip spreading are met, this event is executed. The supernode iterates through all of its neighboring supernodes, and sends each of them the gossip message. The gossip is basically telling the other supernodes that it has registered a high popularity for one video (given a specific search query). This event will generate n instances of `SUPERNODE_RECEIVES_GOSSIP`, where n equals the number of neighbors.

5. `SUPERNODE_RECEIVES_GOSSIP`. Upon receipt of a gossip event from a neighboring supernode, a supernode will check if the gossiped video is present in its own search index. If not, download from the content provider (or the referring supernode). Otherwise, just ignore the gossip.

6. `SUPERNODE_RECALCULATION`. This is an event used in verbose mode, and dumps the entire search index to a logfile.

7. `NEW_CONTENT`. Every now and then, new content is added. What happens then is that a supernode is given a brand new video from the content provider. This new video is added to the supernode's search index, and will participate in the subsequent recommendation calculations. Finally, another instance of `NEW_CONTENT` is queued, based upon the value of the `newContentFrequency` attribute (described in section 4.2.12).

8. `ADD_SUPER_CONTENT`. In test runs, we have defined the term *super content*. This is a video that the simulator administrator manually inserts into the system at a given time. For instance, we can inject a video with popularity 1.0 (which we know that all users will rate as *good*) and monitor the time it takes for the video to replicate itself to all supernodes in the system. A SCARF goal is to minimize this time.

9. `SYSTEM_PROGRESS`. This is an event which is used to implement a progress bar for the simulation. This is used to make it easier for the simulator administrator to watch progress. Whenever an event of this type is executed, another event of the same type is inserted in the queue.

10. `SYSTEM_TERMINATION`. When a simulation is started, one of the configuration parameters is the simulation time. So, when this event type is executed, the system stops and prints out a relevant summary of the simulation.

From the event type name, one can see that three of the events has a user as the *subject* part of the event, three has a supernode as subject, two has content and two has the system itself as subject. Some of the events also has an *object* element. Hence, as seen from a developer the *Event* class has multiple constructors.

For instance, the `SUPERNODE_SPREADS_GOSSIP` event has a subject supernode and up to several object supernodes.

4.2.12 Initialization

Before a simulation can commence, there are several parameters must be specified and tasks must be completed. This is done in this order:

1. **Generate all topics** The vocabulary from which each search query will be fetched is generated.

2. **Generate content** According to the configuration parameter `numberOfVideos` a start-up amount of content is created. These videos are attributed with an identifier, base rank, relevant topics, length and the hidden popularity field.

3. **Generate supernodes and users** Configuration parameters determine how many supernodes and users the system will have. In this step we

generate all supernodes and all users, and finally distribute the recently created content to a subset of the supernodes. The start-up number of videos per supernode is also configurable.

4. **Generate neighborhood** Based on the selected supernode topology, all connections between supernodes are constructed.

5. **Initialize search indexes** All supernodes parse through their dedicated content, and index everything based on topic.

6. **Activate users** The initial user activity is being packed in an event, and put on the event queue. The timeframe between start-up time and when all users shall be active is configurable, and each user enters the event queue at a random time stamp within this timeframe.

7. **Determine termination** An event telling whether the simulation is over is placed in the event queue.

8. **Initialize new content generation** The first content to be inserted after start-up is placed in the event queue. Whenever this event is executed, a similar event is installed.

9. **Initialize the progress bar** For the simulation's usability, it is good to see how the system is progressing until termination.

After these steps, the simulator enters an infinite loop[6] where every part of the system behaves as described in all the previous sections.

[6]Only terminated when the system termination event occurs.

Chapter 5

Testing

By modeling the real world, and converting this model to a simulation, it is imperative to come up with well defined tests and provide means for how to interpret all test results.

In this chapter, we describe the test scenarios used for testing the SCARF simulator.

5.1 Small scale testing

The SCARF simulator has a large number of adjustable parameters. In order to have faith in results from large scale testing, we need to have manually verified execution of small scale scenarios, even if they have no practical meaning for giving interesting results per se.

5.1.1 Test running a small example

The very first test run serves as a proof of concept. That is, we run the system until graceful termination, and carefully follow the evolution. We run the system with quite low settings:

- **numberOfSupernodes** 3 — We run with 3 supernodes.
- **usersPerSupernode** 100 — We have 100 users per supernode, giving us 300 users altogether.
- **availableTopics** `"parsons"`, `"liverpool"`, `"madrugada"`, `"tromsø"` — we limit all searches to come from the vocabulary of these four search terms. That also means that the search indexes are grouped by these four terms.
- **numberOfVideos** 100 — We start out with 100 videos in the content provider. This amount will grow as the simulation runs.
- **numberOfVideosPerSupernode** 40 — Each supernode starts out with a random subset of the videos from the content provider – namely 40 out of 100 videos.

- **newContentFrequenzy** 40 — This means that one of the supernodes will be filled with a new piece of content within the next 40 time units. It will be added to the supernode's search index, and enters the algorithm with similar rules as the rest of the content. Be, of course, aware that this new content will score high on freshness. If it turns out to be an appreciated video among its viewers, then the supernode will sense this trend, and alert all neighboring supernodes about this.

- **whenToTerminate** 1500 — The simulation can in theory run forever, but we want to stop it gracefully after a given time, in this case after 1500 time units.

- **superNodeArchitecture** `RING` — The supernode topology is also configurable. Implemented topologies are: `RING` (one-way ring topology), `DOUBLERING` (two-way ring topology), `TWORINGS` (each supernode is a part of two independent one-way ring topologies), and finally `ALLCONNECTED` (every supernode is fully connected to the other ones).

Running this system takes just a second in real time, and it produces several kilobytes of logs.

Figure 5.1 is a screen shot of the search index for supernode 2 at time 0. We see that the index is based on search topic, and each index element consists of six columns:

- **id** The content identifier.

- **SmartRank** The smart rank for this piece of content. About calculation of smart rank, see section 4.2.5.

- **BaseRank** The start-up ranking, based on traditional ranking algorithms. Note that the SmartRank and BaseRank values are identical at start-up. This is because the supernodes in the beginning have received no feedback from users.

- **Positives** A counter for all positive feedback for the actual video. It starts out as 0.

- **Negatives** A counter for all negative feedback for the actual video. It is also 0 at boot time.

- **Added** The timestamp at which the video was added in this index. Just zeros in the beginning.

Figure 5.2 shows an excerpt of how the search index from figure 5.1 has evolved through 1500 time units. The search index is here sorted by smart rank. If we look at the topic "`liverpool`", we see that the highest ranked video (with `id` 8) has a smart rank of 331. We also see that this supernode has streamed this video to 24 users, and all of them said that it was *good*. Finally, we can see that this video was added at time 476, which means that it was not

```
Here is the entire index for Supernode 2:
  Topic liverpool
    Id  98: SmartRank  84, BaseRank:  84, Positives:  0, Negatives:  0, Added:    0.
    Id  40: SmartRank  76, BaseRank:  76, Positives:  0, Negatives:  0, Added:    0.
    Id  12: SmartRank  75, BaseRank:  75, Positives:  0, Negatives:  0, Added:    0.
    Id  84: SmartRank  31, BaseRank:  31, Positives:  0, Negatives:  0, Added:    0.
    Id  31: SmartRank   6, BaseRank:   6, Positives:  0, Negatives:  0, Added:    0.
  Topic tromsø
    Id  75: SmartRank  89, BaseRank:  89, Positives:  0, Negatives:  0, Added:    0.
    Id  15: SmartRank  88, BaseRank:  88, Positives:  0, Negatives:  0, Added:    0.
    Id   7: SmartRank  74, BaseRank:  74, Positives:  0, Negatives:  0, Added:    0.
    Id  60: SmartRank  70, BaseRank:  70, Positives:  0, Negatives:  0, Added:    0.
    Id  45: SmartRank  62, BaseRank:  62, Positives:  0, Negatives:  0, Added:    0.
    Id   0: SmartRank  61, BaseRank:  61, Positives:  0, Negatives:  0, Added:    0.
    Id  29: SmartRank  61, BaseRank:  61, Positives:  0, Negatives:  0, Added:    0.
    Id  77: SmartRank  41, BaseRank:  41, Positives:  0, Negatives:  0, Added:    0.
    Id  67: SmartRank  34, BaseRank:  34, Positives:  0, Negatives:  0, Added:    0.
    Id  71: SmartRank  13, BaseRank:  13, Positives:  0, Negatives:  0, Added:    0.
    Id  85: SmartRank  11, BaseRank:  11, Positives:  0, Negatives:  0, Added:    0.
    Id   4: SmartRank   8, BaseRank:   8, Positives:  0, Negatives:  0, Added:    0.
    Id  51: SmartRank   6, BaseRank:   6, Positives:  0, Negatives:  0, Added:    0.
  Topic madrugada
    Id  99: SmartRank  64, BaseRank:  64, Positives:  0, Negatives:  0, Added:    0.
    Id  87: SmartRank  63, BaseRank:  63, Positives:  0, Negatives:  0, Added:    0.
    Id  72: SmartRank  40, BaseRank:  40, Positives:  0, Negatives:  0, Added:    0.
    Id  54: SmartRank  40, BaseRank:  40, Positives:  0, Negatives:  0, Added:    0.
    Id  53: SmartRank  35, BaseRank:  35, Positives:  0, Negatives:  0, Added:    0.
    Id  37: SmartRank  30, BaseRank:  30, Positives:  0, Negatives:  0, Added:    0.
    Id  46: SmartRank   7, BaseRank:   7, Positives:  0, Negatives:  0, Added:    0.
    Id  27: SmartRank   4, BaseRank:   4, Positives:  0, Negatives:  0, Added:    0.
  Topic parsons
    Id   5: SmartRank  80, BaseRank:  80, Positives:  0, Negatives:  0, Added:    0.
    Id  65: SmartRank  77, BaseRank:  77, Positives:  0, Negatives:  0, Added:    0.
    Id  22: SmartRank  71, BaseRank:  71, Positives:  0, Negatives:  0, Added:    0.
    Id  17: SmartRank  58, BaseRank:  58, Positives:  0, Negatives:  0, Added:    0.
    Id  18: SmartRank  53, BaseRank:  53, Positives:  0, Negatives:  0, Added:    0.
    Id  32: SmartRank  40, BaseRank:  40, Positives:  0, Negatives:  0, Added:    0.
    Id  93: SmartRank   3, BaseRank:   3, Positives:  0, Negatives:  0, Added:    0.
```

Figure 5.1: *This is how the Search Index look like when the system has run its initialization steps.*

```
Here is the entire index for Supernode 2:
  Topic liverpool
    Id   8: SmartRank 331, BaseRank:  91, Positives: 24, Negatives:  0, Added:  476.
    Id  98: SmartRank 144, BaseRank:  84, Positives: 15, Negatives:  9, Added:    0.
    Id   2: SmartRank  96, BaseRank:  66, Positives: 10, Negatives:  7, Added: 1059.
    Id 159: SmartRank  87, BaseRank:  37, Positives:  7, Negatives:  2, Added: 1151.
    Id 153: SmartRank  61, BaseRank:  61, Positives:  0, Negatives:  0, Added: 1476.
    Id  40: SmartRank  56, BaseRank:  76, Positives: 11, Negatives: 13, Added:    0.
    Id 141: SmartRank  53, BaseRank:  53, Positives:  0, Negatives:  0, Added: 1422.
    Id  12: SmartRank  15, BaseRank:  75, Positives:  9, Negatives: 15, Added:    0.
    Id 155: SmartRank  12, BaseRank:   2, Positives:  1, Negatives:  0, Added: 1074.
    Id 110: SmartRank   3, BaseRank:   3, Positives:  0, Negatives:  0, Added: 1056.
    Id 122: SmartRank   0, BaseRank:   9, Positives:  3, Negatives: 17, Added:  403.
    Id  31: SmartRank   0, BaseRank:   6, Positives:  0, Negatives:  7, Added:    0.
    Id  84: SmartRank   0, BaseRank:  31, Positives:  4, Negatives: 19, Added:    0.
  Topic tromsø
    Id 162: SmartRank 166, BaseRank:  96, Positives: 11, Negatives:  4, Added: 1239.
    Id 117: SmartRank 147, BaseRank:   7, Positives: 15, Negatives:  1, Added:  848.
    Id 112: SmartRank 147, BaseRank:  97, Positives: 15, Negatives: 10, Added:  238.
    Id 129: SmartRank 102, BaseRank:  52, Positives:  5, Negatives:  0, Added: 1122.
    Id  15: SmartRank  98, BaseRank:  88, Positives:  1, Negatives:  0, Added:    0.
    Id 118: SmartRank  93, BaseRank:  93, Positives: 11, Negatives: 11, Added:  319.
    Id 134: SmartRank  81, BaseRank:  11, Positives: 13, Negatives:  6, Added:  689.
    Id   7: SmartRank  74, BaseRank:  74, Positives:  0, Negatives:  0, Added:    0.
    Id  60: SmartRank  70, BaseRank:  70, Positives:  0, Negatives:  0, Added:    0.
    Id 115: SmartRank  69, BaseRank:  89, Positives:  9, Negatives: 11, Added:  250.
    Id  94: SmartRank  67, BaseRank:  67, Positives:  0, Negatives:  0, Added: 1391.
    Id  45: SmartRank  62, BaseRank:  62, Positives:  0, Negatives:  0, Added:    0.
    Id 138: SmartRank  61, BaseRank:  71, Positives:  1, Negatives:  2, Added:  796.
    Id   0: SmartRank  61, BaseRank:  61, Positives:  0, Negatives:  0, Added:    0.
    Id  29: SmartRank  61, BaseRank:  61, Positives:  0, Negatives:  0, Added:    0.
    Id 121: SmartRank  58, BaseRank:  98, Positives:  2, Negatives:  6, Added:  382.
    Id  77: SmartRank  41, BaseRank:  41, Positives:  0, Negatives:  0, Added:    0.
    Id 133: SmartRank  38, BaseRank:  38, Positives:  0, Negatives:  0, Added: 1122.
    Id 127: SmartRank  36, BaseRank:  76, Positives:  2, Negatives:  6, Added:  511.
    Id  67: SmartRank  34, BaseRank:  34, Positives:  0, Negatives:  0, Added:    0.
    Id 119: SmartRank  20, BaseRank:  20, Positives:  0, Negatives:  0, Added:  336.
    Id  71: SmartRank  13, BaseRank:  13, Positives:  0, Negatives:  0, Added:    0.
    Id 168: SmartRank  11, BaseRank:  11, Positives:  0, Negatives:  0, Added: 1384.
    Id  85: SmartRank  11, BaseRank:  11, Positives:  0, Negatives:  0, Added:    0.
    Id 106: SmartRank  10, BaseRank:  10, Positives:  0, Negatives:  0, Added:  140.
    Id  75: SmartRank  10, BaseRank:  89, Positives:  3, Negatives: 22, Added:    0.
    Id   4: SmartRank   8, BaseRank:   8, Positives:  0, Negatives:  0, Added:    0.
    Id  51: SmartRank   6, BaseRank:   6, Positives:  0, Negatives:  0, Added:    0.
    Id 130: SmartRank   0, BaseRank:   2, Positives:  0, Negatives:  1, Added:  610.
    Id 116: SmartRank   0, BaseRank:  21, Positives:  0, Negatives:  3, Added:  282.
  Topic madrugada
    Id 100: SmartRank 380, BaseRank: 100, Positives: 28, Negatives:  0, Added:  575.
```

Figure 5.2: *This is how the Search Index look like after running 1500 time units.*

present in this supernode's store at start-up time. Hence, the neighbor[1] must have spread gossip about it based on feedback from own users during the first 476 time units.

If we look at the search index for the topic "`tromsø`", we see on second place a video that was added at time 848 (it has `id` 117). It had a very low base rank (7), but it turned out to be a very popular video after all. The reason that users were recommended this video in the first place, despite the low base rank, is that it in the time after 848 was very highly ranked on the freshness axis.

Finally, inspecting the rest of this figure (and the other logs) indicates that the simulator works as expected.

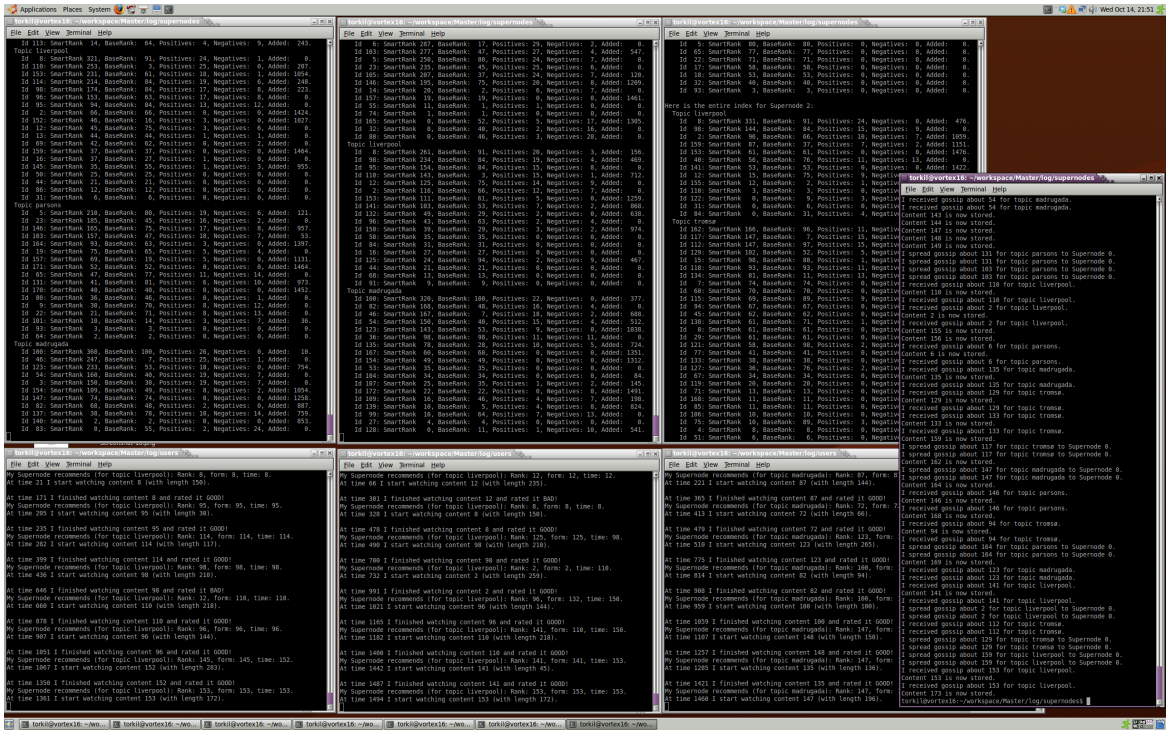

Figure 5.3: *This is a scaled down screen shot on how one of the screens are setup to constantly monitor simulation test runs. It is included here just to serve as an indicator on the amount of concurrent things going on.*

5.2 Large scale testing

The execution of the small scale testing indicates that all elements of the simulation works as planned. Now, we continue with more realistic simulations to extract knowledge of behavior.

One of the main goals of SCARF is to make popular content replicated to as many supernodes as possible, and as quickly as possible. What we have tested is

[1]Since this initial test run was executed in a `RING` topology, all supernodes have only one neighbor.

the insertion of popular content at a given time, and then monitored a) how many supernodes it spreads to, and b) how quickly this was achieved.

Altogether 200 test executions have been run, and this is the applied configuration:

- **numberOfSupernodes** 20
- **usersPerSupernode** 100 — We have 100 users per supernode, which means 2000 users running in parallel.
- **availableTopics** `"parsons"`, `"liverpool"`, `"madrugada"`, `"tromsø"` — a vocabulary of four search terms.
- **numberOfVideos** 100
- **numberOfVideosPerSupernode** 40
- **newContentFrequenzy** 40
- **whenToTerminate** 10000

Two parameters were tuned:

- **superNodeArchitecture** — There were executed 50 test runs for each of the four architectures.
- **realPopularity** — For each architecture, we executed test runs for content of `realPopularity` 1.0, content of popularity 0.9, and similar for 0.8, 0.7 and 0.6.

Each test configuration was executed 10 times. The content that we monitor was for every test inserted exactly at time 10.

5.2.1 The `RING` topology

In the `RING` topology, each supernode is attached to exactly one other. Hence, for content that resides on one supernode, it must pass 19 steps[2] of gossiping before it has reached full cumulative bandwidth. For content of popularity 1.0, it can be assumed that it will spread a lot faster than content with popularity 0.6. The latter might not even pass all 19 steps of gossip-ping.

[2] given 20 supernodes.

Let us inspect the test results:

popularity	1.0	0.9	0.8	0.7	0.6
test run 1	3720	3253	−16	−4	−2
test run 2	3228	−11	−4	−2	−1
test run 3	3284	2924	−7	−5	−1
test run 4	3468	4136	−6	−1	−1
test run 5	3520	−2	−4	−6	−4
test run 6	3047	3966	−17	−6	−1
test run 7	3067	−9	−8	−1	−1
test run 8	3394	3640	−5	−5	−1
test run 9	7184	3921	−9	−2	−2
test run 10	3592	3770	−8	−1	−1

If the table entry is a positive integer, it indicates the number of time units it took for the injected content to reach all 20 supernodes. If the entry is a negative integer, it did not reach all supernodes. The negative number itself is a number for how many supernodes it managed to spread to during the simulation. As an example, see test run 2 for content of popularity `0.9`. Its entry says "-11", which means two things: a) we could not measure the time to spread to all 20 supernodes, because it did not, and b) the content has replicated itself to 11 supernodes in the system.

Figure 5.4 shows output generated by the 10th test run where the injected content had popularity `1.0`. It is easy to follow the distribution by looking at the timestamp for when the content was being present on the different supernodes.

```
################ Created ################
At time   10 I am served by Supernode  0 ( 1)
At time  377 I am served by Supernode  1 ( 2)
At time  675 I am served by Supernode  2 ( 3)
At time  812 I am served by Supernode  3 ( 4)
At time  954 I am served by Supernode  4 ( 5)
At time 1114 I am served by Supernode  5 ( 6)
At time 1248 I am served by Supernode  6 ( 7)
At time 1373 I am served by Supernode  7 ( 8)
At time 1643 I am served by Supernode  8 ( 9)
At time 1853 I am served by Supernode  9 (10)
At time 2146 I am served by Supernode 10 (11)
At time 2290 I am served by Supernode 11 (12)
At time 2497 I am served by Supernode 12 (13)
At time 2690 I am served by Supernode 13 (14)
At time 2835 I am served by Supernode 14 (15)
At time 3011 I am served by Supernode 15 (16)
At time 3243 I am served by Supernode 16 (17)
At time 3357 I am served by Supernode 17 (18)
At time 3474 I am served by Supernode 18 (19)
At time 3592 I am served by Supernode 19 (20)
```

Figure 5.4: *Here is the output from the 10th test run where the injected content had popularity `1.0`, and with the supernodes in a `RING` topology.*

However, looking at the similar log for tracking the distribution of a video

with popularity `0.8`, we see that it eventually reached a supernode where the users did not find it good enough to trigger a gossip message to the next supernode. Look at figure 5.5, which shows the tracks for the 10th test run for a video with popularity `0.8` (from the table above).

```
################# Created #################
At time   10 I am served by Supernode  0 ( 1)
At time  151 I am served by Supernode  1 ( 2)
At time  326 I am served by Supernode  2 ( 3)
At time  502 I am served by Supernode  3 ( 4)
At time  627 I am served by Supernode  4 ( 5)
At time  808 I am served by Supernode  5 ( 6)
At time 1025 I am served by Supernode  6 ( 7)
At time 1348 I am served by Supernode  7 ( 8)
```

Figure 5.5: *Here is the output from the 10th test run, where the injected content had popularity `0.8`, and with the supernodes in a `RING` topology.*

To derive how fast and far the injected content reached, we can average over the numbers:

popularity	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
avg time to reach all	3350	3659	*N/A*	*N/A*	*N/A*
number of nodes reached	20.0	16.2	8.4	3.4	1.5

We see that only content of popularity `0.9` and `1.0` reached every supernode, and we see that the higher popularity, the faster spreading. Similarly, we see that the higher popularity, the higher number of supernodes are reached by the injected content.

These results are conform with the goal of SCARF.

5.2.2 The `DOUBLERING` topology

So, how would SCARF behave if we change the supernode topology to a double ring? In this topology, the supernodes are still connected in a ring, but now gossip is spread in both directions in the ring.

We did the same 50 tests for this topology, and got these results:

popularity	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
test run 1	2190	2170	3788	−4	−3
test run 2	1823	2421	−7	−1	−1
test run 3	1860	1788	−7	−1	−4
test run 4	1927	2047	−1	−6	−3
test run 5	2027	1866	−4	−6	−4
test run 6	1728	2139	−6	−1	−3
test run 7	1808	2035	1831	−11	−5
test run 8	1850	2792	−16	−7	−1
test run 9	2023	4200	−15	−11	−1
test run 10	1623	1995	2426	−4	−1

Compared with the results from the `RING` topology, we see that the real

popular content reached full cumulative bandwidth significantly faster. We also see that content with popularity `0.8` here was able to achieve full cumulative bandwidth for three runs.

See figure 5.6 for the tracks of the 10th run for a video with popularity `1.0`. By looking at the supernode ids, it is shown that the content is spread both ways in the ring.

```
################ Created #################
At time   10 I am served by Supernode  0 ( 1)
At time  210 I am served by Supernode  1 ( 2)
At time  210 I am served by Supernode 19 ( 3)
At time  334 I am served by Supernode 18 ( 4)
At time  401 I am served by Supernode  2 ( 5)
At time  524 I am served by Supernode 17 ( 6)
At time  551 I am served by Supernode  3 ( 7)
At time  656 I am served by Supernode 16 ( 8)
At time  676 I am served by Supernode  4 ( 9)
At time  784 I am served by Supernode 15 (10)
At time  878 I am served by Supernode  5 (11)
At time  924 I am served by Supernode 14 (12)
At time 1115 I am served by Supernode 13 (13)
At time 1125 I am served by Supernode  6 (14)
At time 1266 I am served by Supernode  7 (15)
At time 1291 I am served by Supernode 12 (16)
At time 1394 I am served by Supernode  8 (17)
At time 1430 I am served by Supernode 11 (18)
At time 1615 I am served by Supernode  9 (19)
At time 1623 I am served by Supernode 10 (20)
```

Figure 5.6: *Here is the output from the 10th test run, where the injected content had popularity `1.0`, and with the supernodes in a `DOUBLERING` topology.*

Here are the average numbers from this table:

popularity	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
avg time to reach all	1886	2345	2681	*N/A*	*N/A*
number of nodes reached	20.0	20.0	11.6	5.2	2.8

These numbers confirm that popular content reached all supernodes faster. It is also noteworthy that videos of popularity `0.9` reached all supernodes for every test run with this topology. This did not happen when executing the tests in the `RING` topology.

5.2.3 The `TWORINGS` topology

Will we see other behavior if we employ a topology where the supernodes are connected in two independent rings? Or will the behavior be the same as for the `DOUBLERING`?

These are the numbers:

popularity	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
test run 1	1807	1818	−10	2762	−1
test run 2	1723	1896	−9	−5	−1
test run 3	1895	1775	−1	−9	−3
test run 4	1850	2060	−8	−1	−1
test run 5	2069	2219	−1	−1	−1
test run 6	1767	2006	2911	−9	−9
test run 7	1675	1753	−3	−6	−1
test run 8	1868	2021	1900	−15	−1
test run 9	1806	2264	−19	−5	−3
test run 10	1653	2127	1952	−4	−5

Comparing the execution results from the `DOUBLERING` and the `TWORINGS` topology indicates that the system behavior is more or less identical. The only obvious difference is that one of the executions with a video of popularity `0.7` actually reached every supernode in the `TWORINGS` topology. However, one single occurrence does not provide significant results to state a conclusion.

See figure 5.7 for the tracks of the 10th run for a video with popularity `1.0`. Note how the content is spread along two different rings, by watching the supernode ids.

```
################# Created ##################
At time   10 I am served by Supernode  0 ( 1)
At time  154 I am served by Supernode  1 ( 2)
At time  154 I am served by Supernode 10 ( 3)
At time  285 I am served by Supernode 11 ( 4)
At time  291 I am served by Supernode  2 ( 5)
At time  425 I am served by Supernode  3 ( 6)
At time  425 I am served by Supernode 12 ( 7)
At time  552 I am served by Supernode 13 ( 8)
At time  701 I am served by Supernode 14 ( 9)
At time  730 I am served by Supernode  4 (10)
At time  831 I am served by Supernode 15 (11)
At time  878 I am served by Supernode  5 (12)
At time 1017 I am served by Supernode 16 (13)
At time 1151 I am served by Supernode  6 (14)
At time 1156 I am served by Supernode 17 (15)
At time 1285 I am served by Supernode 18 (16)
At time 1285 I am served by Supernode  7 (17)
At time 1429 I am served by Supernode  8 (18)
At time 1477 I am served by Supernode 19 (19)
At time 1653 I am served by Supernode  9 (20)
```

Figure 5.7: *Here is the output from the 10th test run, where the injected content had popularity `1.0`, and with the supernodes in a `TWORINGS` topology.*

Here are the average numbers:

popularity	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
avg time to reach all	1811	1993	2354	1762	N/A
number of nodes reached	20.0	20.0	11.1	7.5	2.6

We see that the videos reach all supernodes slightly faster in a `TWORINGS`

topology than in a `DOUBLERING`. However, this deviation might very well be caused by several occurrences of randomness within the entire system. The number of reached supernodes is practically the same for the two topologies.

5.2.4 The `ALLCONNECTED` topology

For the final 50 test runs, we connect all supernodes with each other. A consequence of that is that whenever a supernode triggers gossip, all supernodes will be reached simultaneously.

Here are the test results:

popularity	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
test run 1	169	311	148	159	−1
test run 2	366	302	265	328	403
test run 3	341	318	179	−1	−1
test run 4	325	279	179	250	−1
test run 5	311	224	−1	429	−1
test run 6	172	194	260	192	−1
test run 7	303	187	234	−1	−1
test run 8	369	318	−1	298	−1
test run 9	281	335	359	159	359
test run 10	177	328	285	232	−1

These results are very different from the previous. We see that every content basically is interpreted as *good* or *bad*. And all results are actually made on the supernode on which the content initially was inserted. If it on that supernode turns out to be so good that it triggers gossip, it ends up on the entire system. And this goes really fast. If the gossip trigger criteria are not met, then none of the other supernodes are getting aware of its existence at all.

Look at the averages:

popularity	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
avg time to reach all	281	280	239	256	331
number of nodes reached	20.0	20.0	16.2	16.2	4.8

Actually, the content with popularity `0.8` was in these test runs the quickest to reach all supernodes. This is merely because it was popular enough to trigger the gossip criteria at the same time as users chose to watch this video quicker than they chose to watch the ones with popularity `1.0` and `0.9`. That choice carries some randomness, and the users never know the actual popularity variable.

See figure 5.8 for the output of the 10th test execution with a fully connected supernode topology, injected with a video with popularity `0.9`. At time 328, we see that enough users at supernode 0 have watched the video, given it a good rating, and hence triggered a gossip from the supernode. As the protocol says that a supernode always will include the videos about which they receive gossip, all supernodes add this to their local indexes.

```
################## Created ##################
At time   10 I am served by Supernode  0 ( 1)
At time  328 I am served by Supernode  1 ( 2)
At time  328 I am served by Supernode  2 ( 3)
At time  328 I am served by Supernode  3 ( 4)
At time  328 I am served by Supernode  4 ( 5)
At time  328 I am served by Supernode  5 ( 6)
At time  328 I am served by Supernode  6 ( 7)
At time  328 I am served by Supernode  7 ( 8)
At time  328 I am served by Supernode  8 ( 9)
At time  328 I am served by Supernode  9 (10)
At time  328 I am served by Supernode 10 (11)
At time  328 I am served by Supernode 11 (12)
At time  328 I am served by Supernode 12 (13)
At time  328 I am served by Supernode 13 (14)
At time  328 I am served by Supernode 14 (15)
At time  328 I am served by Supernode 15 (16)
At time  328 I am served by Supernode 16 (17)
At time  328 I am served by Supernode 17 (18)
At time  328 I am served by Supernode 18 (19)
At time  328 I am served by Supernode 19 (20)
```

Figure 5.8: *Here is the output from the 10th test run, where the injected content had popularity `0.9`, and with the supernodes in an `ALLCONNECTED` topology.*

5.3 Discussion

The most important conclusion we can draw after running these simulations, is that the main goal is achieved: Popular content is replicated throughout the system, and the distribution speed is usually dependent on the popularity of the content.

Figure 5.9 shows how fast the distribution throughout the system went for the different topologies. The y-axis shows time. At time 10, an item with popularity 1.0 is inserted on one supernode. The x-axis tells how many nodes on which this item is present. All four topologies are shown.

In a real world installation, a decision on which supernode topology to employ must be taken, and we see that depending on the system's requirements and limitations, different topologies can be chosen. Perhaps running other topologies than implemented in this SCARF simulation might give even better results. There are numerous of topologies from which to choose. *Cube* topologies of n dimensions could be tried, with various values of n. Also, sending the gossip message to a random number of random other supernodes might give interesting results.

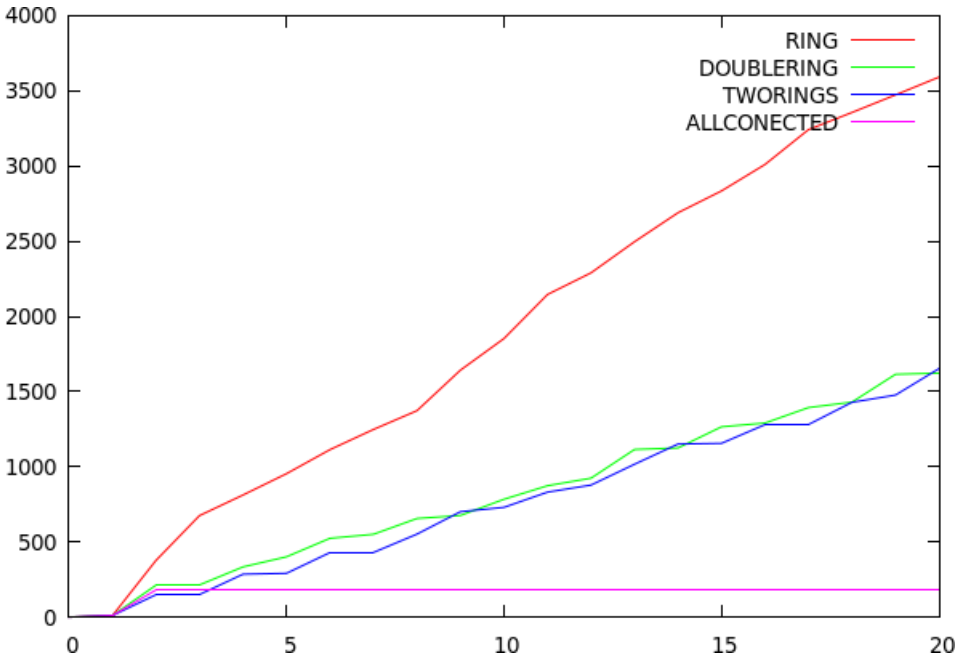

Figure 5.9: *This graph shows how fast content with popularity* 1.0 *was distributed throughout the supernodes.*

Chapter 6

Conclusion

6.1 What is achieved?

If a good Internet multimedia service turns out to be successful, and attract a large number of users, there are basically just two ways to allow tremendous growth:

1. Throw in piles of money to buy all needed server power and bandwidth.
2. Be smart enough with the design and deployment.

The most successful actors (like Google [15], YouTube [45], etc.) employ combinations of these two alternatives. In academia, we prefer to search for solutions for scalable problems in the second category.

In SCARF, it has been shown that the algorithms applied for popular content to spread across the network makes it possible to serve a huge number of users with a high percentage of popular content. This has been done by combining a replication strategy, user feedback and a recommendation engine.

By looking at the problem definition in section 1.2, we identified four tasks that SCARF were supposed to solve.

1. **The architecture** — We have designed and implemented a simulation for an architecture that offloads work from a large content provider. In addition, several topologies for the caching layer have been discussed and implemented.
2. **The feedback** — SCARF allows users to provide positive or negative feedback to all videos, and the design of the supernodes assures that the feedback is collected and applied for the ranking algorithm.
3. **The recommendation engine** — In SCARF recommendation is done along three different axes. These axes covers 1) a "smart" ranking, 2) up-and-coming trendy videos, and 3) fresh material.
4. **The availability** — Test results indicate that SCARF assures that the time it takes to maximize availability for very popular content is reduced.

We have throughout the thesis discussed some assumptions and simplifications. Still we believe that SCARF faithfully and realistically model the most important aspects of a real implementation and deployment.

6.2 Future work

The most obvious thing to do to expand SCARF would be to implement and deploy such a system in a real world setting. That requires a lot of money, machines, locations, time, and users.

The real world deployment is unlikely, but there are still related theory that could be interesting to dilate. In the SCARF implementation, a user will not be able to find content on other servers than already present on the supernode to which it belongs. The supernode does its best to provide all the interesting media, but it would be very interesting to devise an architecture and protocols that assure that all supernodes have an index of *all available content* from the content provider. The users should be allowed to have an evanescent relationship to its supernode, and schemes for roaming between supernodes could also be developed.

The SCARF system is also built upon optimistic strategies, where a general assumption on that everything works as planned is taken. To make this system deployable, there are several fault-tolerance aspects that need to be solved. The system is especially vulnerable if supernodes should be disconnected from the network. In such cases, there needs to be emergency protocols for a) how to redistribute the supernode's end users to other supernodes, and b) how to regenerate the internal supernode network in accordance with the applied supernode network topology.

Finally, some of the parameters used to configure a SCARF simulation are set based on assumptions and simplifications. Examples of parameters that could be tuned further are the popularity distribution, the gossip trigger, the definition of trendy videos, and the smart rank algorithm. More research in tuning these parameters might produce valuable results.

Bibliography

[1] Reid Andersen, Christian Borgs, Jennifer Chayes, Uriel Feige, Abraham Flaxman, Adam Kalai, Vahab Mirrokni, and Moshe Tennenholtz. Trust-based recommendation systems: an axiomatic approach. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 199–208, New York, NY, USA, 2008. ACM.

[2] Scott M. Baker and Bongki Moon. Distributed cooperative Web servers. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 31:1215–1229, 1999.

[3] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Commun. ACM*, 46(2):43–48, 2003.

[4] Heinz Breu, Joseph Gil, David Kirkpatrick, and Michael Werman. Linear time euclidean distance transform algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:529–533, 1995.

[5] Robin Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, 2002.

[6] The Free Dictionary by Farlex. `http://encyclopedia2.thefreedictionary.com/`.

[7] Bengt Carlsson and Rune Gustavsson. The rise and fall of napster - an evolutionary approach. In *AMT '01: Proceedings of the 6th International Computer Science Conference on Active Media Technology*, pages 347–354, London, UK, 2001. Springer-Verlag.

[8] Ian Clarke, S. Oskar, On Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *In Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[9] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Computer*, 22(2):63–70, 1989.

[10] UC San Diego. `http://www.ucsd.edu/`.

[11] Herb Edelstein. Unraveling Cilent/Server Architecture. *DBMS 7*, May 1994.

[12] WAIF (Wide Area Information Filtering).
`http://www.waif.cs.uit.no/`.

[13] Masoumeh Ghahremani, Seyed-Amin Hosseini-Seno, and Rahmat Budiarto. A new approach for web applications examination before publishing. *Future Computer and Communication, International Conference on*, 0:470–473, 2009.

[14] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, 1992.

[15] Google.
`http://www.google.com/`.

[16] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. Does internet media traffic really follow zipf-like distribution? In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 359–360, New York, NY, USA, 2007. ACM.

[17] Daniel E. Hastings. Telecoms convergence consortium.

[18] James O. Henriksen, Robert M. O'Keefe, C. Dennis Pegden, Robert G. Sargent, and Brian W. Unger. Implementations of time (panel). In Douglas W. Jones, editor, *WSC '86: Proceedings of the 18th conference on Winter simulation*, pages 409–416, New York, NY, USA, 1986. ACM.

[19] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[20] Shiu-li Huang. Comparision of utility-based recommendation methods. *PACIS 2008 Proceedings*, Paper 21, 2008.

[21] Carnegie Mellom Software Engineering Institute. Client/Server Software Architectures–An Overview.
`http://www.sei.cmu.edu/str/descriptions/clientserver`.

[22] Márk Jelasity and Ozalp Babaoglu. T-Man: Gossip-based overlay topology management. In *Proceedings of Engineering Self-Organising Applications (ESOA'05)*, July 2005.

[23] M. Kaashoek and D. Karger. Koorde: A simple degree-optimal distributed hash table, 2003.

[24] Bruce Krulwich. Lifestyle finder: Intelligent user profiling using large-scale demographic data. *AI Magazine*, 18(2):37–45, 1997.

[25] Bruce Krulwich and Chad Burkey. The infofinder agent: Learning user interests through heuristic phrase extraction. *IEEE Intelligent Systems*, 12(5):22–27, 1997.

[26] Last.fm.
`http://last.fm/`.

[27] Virginia Lo, Dayi Zhou, Yuhong Liu, Chris Gauthierdickey, and Jun Li. Scalable supernode selection in peer-to-peer overlay networks. In *In Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems, La*, pages 18–25. IEEE, 2005.

[28] Luiz R. Monnerat and Claudio L. Amorim. D1ht: A distributed one hop hash table. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 10 pp.+, April 2006.

[29] M. E. J. Newman. Power laws, pareto distributions and zipf's law, December 2004.

[30] Aviv Nisgav and Boaz Patt-Shamir. Finding similar users in social networks. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 169–177, New York, NY, USA, 2009. ACM.

[31] Douglas W. Oard and Jinmook Kim. Modeling information content using observable behavior, 2001.

[32] University of Tromsø.
`http://www.uit.no/`.

[33] Sylvia Ratnasamy, Paul Francis, Scott Shenker, and Mark Handley. A scalable content-addressable network. In *In Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[34] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system. 2002.

[35] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001.

[36] George Schussel. Client/Server Past, Present, and Future [online].
`http://news.dci.com/geos/dbsejava.htm`, (September, 2008).

[37] A. Shepitsen, J. Gemmell, B. Mobasher, and R. Burke. Personalized recommendation in social tagging systems using hierarchical clustering. In *Proceedings of the 2008 ACM Conference on Recommender Systems (RecSys 2008)*, pages 259–266, New York, NY, USA, October 2008. ACM.

[38] Xiaowei Shi. An intelligent knowledge-based recommendation system. pages 431–435, 2005.

[39] Spotify.
`http://www.spotify.com/`.

[40] Ion Stoica, Robert Morris, David Karger, Frans M. Kaashoek, and Hari. Chord: A scalable peer-to-peer lookup service for internet applications, 2001.

[41] Loren Terveen, Will Hill, Brian Amento, David Mcdonald, and Josh Creter. Phoaks: A system for sharing recommendations. *Communications of the ACM*, 40(3):59–62, 1997.

[42] New York Times.
`http://www.nytimes.com/`.

[43] Cornell University.
`http://www.cornell.edu/`.

[44] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[45] YouTube.
`http://www.youtube.com/`.

[46] Xi-Zheng Zhang. Building personalized recommendation system in e-commerce using association rule-based mining and classification. In *Machine Learning and Cybernetics, 2007 International Conference on*, volume 7, pages 4113–4118, 2007.

[47] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2004.

[48] Philip Zigoris and Yi Zhang. Bayesian adaptive user profiling with explicit & implicit feedback. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 397–404, New York, NY, USA, 2006. ACM.

Appendix A

Source file listings

A.1 Source code

A.1.1 Simulation.java

```
package master;

import master.User;
import master.Supernode;
import master.Topic;
import master.Event;

import java.io.FileWriter;
import java.io.IOException;
import java.util.Collections;
import java.util.Comparator;
import java.util.Random;
import java.util.ArrayList;
import java.util.PriorityQueue;

/**
 * Simulation.java
 * Purpose: This is the main class for running the entire simulator.
 *          It initializes all elements of the system, and runs until
    termination.
 *
 * @author Torkil Grindstein
 * @version 1.0
 *
 */
public class Simulation {

  static Supernode[] supernodes;
  static User[] users;
  static ArrayList<Content> contents = new ArrayList<Content>();
  static Topic[] topics;
  static boolean DEBUGMODE = false;
  static enum superNodeArchitecture {RING, DOUBLERING, TWORINGS,
      ALLCONNECTED};

  static PriorityQueue<Event> eventQueue = new PriorityQueue<Event>();

  static String logDirectory = "/home/torkil/workspace/Master/log/
      measures/";
  static String logFile = logDirectory + "Simulation.log";

  /**
```

```
40     * @param args
       */
      public static void main(String[] args) {
        System.out.println("I'm running!");

        // Generate n Supernodes, each with m Users. First, set some
            constants
        final int numberOfSupernodes = 200;
        final int usersPerSupernode = 200;
        // final String[] availableTopics = {"parsons", "liverpool", "
            madrugada", "tromsÃÿ", "sortland", "hamna", "flodhest", "
            fÃěrikÃěl"};
        final String[] availableTopics = {"parsons", "liverpool", "madrugada"
            , "tromsÃÿ"};
50      // final String[] availableTopics = {"parsons", "liverpool"};
        final int numberOfVideos = 100;
        final int numberOfVideosPerSupernode = 40;
        final int newContentFrequenzy = 40;       // The range from within new
             content will be added
        final int whenToTerminate = 10000;
        final superNodeArchitecture architecture = superNodeArchitecture.
            ALLCONNECTED;

        supernodes = new Supernode[numberOfSupernodes];
        InitializeSystem(usersPerSupernode, availableTopics, numberOfVideos,
            numberOfVideosPerSupernode, architecture);
        if (DEBUGMODE) DebugPrintSizes();
60
        // Build each supernode's index
        for (Supernode s: supernodes) {
          BuildIndex(s);
        }

        Random generator = new Random();

        InitializeUsers();

70      // Decide when to stop (to prevent running wild...)
        Event terminationEvent = new Event(whenToTerminate, "System", 0,
            Event.eventType.SYSTEM_TERMINATION, -1);
        eventQueue.add(terminationEvent);

        // Let new content appear (Make sure it arrives after the super
            content
        Event newContentEvent = new Event(12+generator.nextInt(
            newContentFrequenzy), "Content", -1, Event.eventType.NEW_CONTENT,
             -1);
        eventQueue.add(newContentEvent);

        // OK, here comes some specially inserted data, used for monitoring
            and performance measurement
        // At time 10 insert a super video on Supernode 1. When will it be
            available on every Supernode?
80      Event superContentEvent = new Event(10, "Content", -1, Event.
            eventType.ADD_SUPER_CONTENT, -1);
        eventQueue.add(superContentEvent);

        // Print progress bar...
        Event progressEvent = new Event(0, "System", 1, Event.eventType.
            SYSTEM_PROGRESS, -1);
        eventQueue.add(progressEvent);

        // Write out cumulative bandwidth stats every 100 sec
  //      if (time % 100 == 0) {
  //        logCumulativeBandwidth(time);
90 //      }
```

```
// Let the game begin...
while (true) {
  int time;
  Event event;
  while (!eventQueue.isEmpty()) {
    // Pick the closest event:
    event = eventQueue.poll();
    time = event.timestamp;
    switch (event.eventName) {
    case USER_STARTS_WATCHING:
      if (event.objectId < 0) { // The Supernode could not recommend
          - try again!
        Recommendation rec =  supernodes[users[event.actorId].
            supernodeServer].recommend(users[event.actorId].
            topicInterest, users[event.actorId].haveSeen);
        if (DEBUGMODE) users[event.actorId].logRecommendation(rec);
        int newContentToWatch = rec.getRandomRecommendation();
        Event newEvent = new Event(time + generator.nextInt(30), "
            User", event.actorId, Event.eventType.
            USER_STARTS_WATCHING, newContentToWatch);
        eventQueue.add(newEvent);
      } else {
        // Watch video...
        users[event.actorId].currentlyWatching = contents.get(event.
            objectId);
        users[event.actorId].timeStartedWatching = time;
        if (DEBUGMODE) users[event.actorId].logStartsWatching(time,
            event.objectId, contents.get(event.objectId).length);
        // Set end time.
        Event newEvent = new Event(time + contents.get(event.objectId
            ).length, "User", event.actorId, Event.eventType.
            USER_STOPS_WATCHING, event.objectId);
        eventQueue.add(newEvent);
      }
      break;
    case USER_IDLE:
      // Time to watch a new video...
      Recommendation rec =  supernodes[users[event.actorId].
          supernodeServer].recommend(users[event.actorId].
          topicInterest, users[event.actorId].haveSeen);
      int newContentToWatch = rec.getRandomRecommendation();
      if (newContentToWatch < 0) { // The supernode could not find
          anything appropriate
        if (DEBUGMODE) users[event.actorId].logNothingAvailable(time)
            ;
        Event nothingAvailableEvent = new Event(time + generator.
            nextInt(60), "User", event.actorId, Event.eventType.
            USER_IDLE, -1);
        eventQueue.add(nothingAvailableEvent);
      } else {
        if (DEBUGMODE) users[event.actorId].logRecommendation(rec);
        Event newVideoEvent = new Event(time + generator.nextInt(30),
             "User", event.actorId, Event.eventType.
            USER_STARTS_WATCHING, newContentToWatch);
        eventQueue.add(newVideoEvent);
      }
      break;
    case USER_STOPS_WATCHING:
      // Video is finished. Rate the video!
      Topic t = users[event.actorId].currentlyWatching.relevantTopics
          [0];
      Content c = users[event.actorId].currentlyWatching;
      String rate = "";

      // Update index
```

```
            for (IndexEntry ind_entry: supernodes[users[event.actorId].
                supernodeServer].index) {
              for (Topic topic: ind_entry.entry.keySet()) {
                if (topic.equals(t)) {
                  for (IndexElement ie: ind_entry.entry.get(topic)) {
                    if (ie.contentId == c.contentId) {
                      // Found right index element, now update ranks/
                          counters
                      if (users[event.actorId].currentlyWatching.
                          realPopularity > generator.nextDouble()) {
                        ie.numberOfPositives++;
                        ie.smartRank += 10; // This could be tuned
                        rate = "GOOD!";
                        // Decide whether this positive ranking should lead
                            to a gossip message to the neighbors
                        if (ie.numberOfPositives == 5 && ie.
                            numberOfNegatives <= 2) { // This could be
                            tuned
                          Event gossipEvent = new Event(time, "Supernode",
                              users[event.actorId].supernodeServer, Event.
                              eventType.SUPERNODE_SPREADS_GOSSIP, c, t);
                          eventQueue.add(gossipEvent);
                        }
                      } else {
                        ie.numberOfNegatives++;
                        ie.smartRank = Math.max(0, ie.smartRank-10);
                        rate = "BAD!";
                      }
                      if (DEBUGMODE) {
                        users[event.actorId].logFinishedWatching(time, c.
                            contentId, rate);
                        Event supernodeEvent = new Event(time, "Supernode",
                             users[event.actorId].supernodeServer, Event.
                            eventType.SUPERNODE_RECALCULATION, -1);
                        if (!eventQueue.contains(supernodeEvent)) {
                          eventQueue.add(supernodeEvent);
                        }
                      }
                    }
                  }
                }
              }
            }

            users[event.actorId].justSawContent(users[event.actorId].
                currentlyWatching);
            users[event.actorId].currentlyWatching = null;

            // Idle a bit before starting watching a new video
            Event userIdleEvent = new Event (time + generator.nextInt(30),
                "User", event.actorId, Event.eventType.USER_IDLE, -1);
            eventQueue.add(userIdleEvent);
            break;
          case SUPERNODE_RECALCULATION:
            supernodes[event.actorId].logEntireIndex();
            break;
          case SUPERNODE_SPREADS_GOSSIP:
            supernodes[event.actorId].spreadGossip(time, event.content,
                event.topic);
            for (Supernode neighbor: supernodes[event.actorId].neighbors) {
              Event receiveEvent = new Event(time+1, "Supernode", neighbor.
                  supernodeId, Event.eventType.SUPERNODE_RECEIVES_GOSSIP,
                  -1);
              eventQueue.add(receiveEvent);
            }
            break;
```

```
            case SUPERNODE_RECEIVES_GOSSIP:
190           supernodes[event.actorId].receiveGossip(time);
              break;
            case NEW_CONTENT:
              Topic relatedTopic = topics[generator.nextInt(topics.length)];
              Topic[] relatedTopics = new Topic[1];
              relatedTopics[0] = relatedTopic;
              int contentId = contents.size();
              Content newContent = new Content(
                  contentId,                    //
                  generator.nextInt(100),       // BaseRank is in [0, 99] ->
                      99 highest rank
200               relatedTopics,                // Currently only relevant for
                       one topic
                  generator.nextInt(291) + 10   // Assume length is in [10,
                      300]
              );
              contents.add(newContent);
              // Add to a supernode
              int sn = generator.nextInt(supernodes.length);
              supernodes[sn].addContent(newContent, time);
              // Add to index
              IndexElement ie = new IndexElement(newContent.contentId,
                  newContent.baseRank, time);
              supernodes[sn].addIndexElementToIndex(newContent.relevantTopics
                  [0], ie);
210
              // Let yet another video be available...
              Event moreNewContentEvent = new Event(time + generator.nextInt(
                  newContentFrequenzy), "Content", -1, Event.eventType.
                  NEW_CONTENT, -1);
              eventQueue.add(moreNewContentEvent);
              break;
            case ADD_SUPER_CONTENT:
              Topic relTopic = topics[generator.nextInt(topics.length)];
              Topic[] relTopics = new Topic[1];
              relTopics[0] = relTopic;
              int cId = contents.size();
220           Content superContent = new Content(
                    cId,              //
                    100,              //
                    relTopics,        //
                    100,              // Length is 100
                    1.0,              // Super popularity
                    true              // Super content
                  );
              contents.add(superContent);
              logSuperContentCreation(time, superContent, 0);
230           supernodes[0].addContent(superContent, time);
              IndexElement i = new IndexElement(superContent.contentId,
                  superContent.baseRank, time);
              supernodes[0].addIndexElementToIndex(superContent.
                  relevantTopics[0], i);
              break;
            case SYSTEM_PROGRESS:
              if (event.actorId == 1) {
                System.out.println("\nProgress...");
                System.out.println("
                    ==========================================================
                    ");
              } else {
                System.out.print("|");
240           }
              int progressSlot = whenToTerminate / 100;
              Event nextProgressSlotEvent = new Event (event.timestamp +
                  progressSlot, "System", -1, Event.eventType.SYSTEM_PROGRESS
```

```
            , -1);
          eventQueue.add(nextProgressSlotEvent);
          break;
        case SYSTEM_TERMINATION:
          for (Supernode s: supernodes) {
            s.logEntireIndex();
          }
          System.out.println("\nThe system successfully ends after " +
              event.timestamp + " seconds.");
          System.exit(0);

          // Should not be here...
        default:
          System.out.println("Terminating due to unknown event: " + event
              .eventName.toString());
          System.exit(0);
        }
      }
    }

    // Determine when each User should start watching.
    // Current algorithm selects that user should be active within 30
       seconds, uniformly
    private static void InitializeUsers() {
      Random generator = new Random();
      for (User u: users) {
        // Determine which video to watch
        Recommendation rec =  supernodes[users[u.userId].supernodeServer].
            recommend(users[u.userId].topicInterest, users[u.userId].
            haveSeen);
        if (DEBUGMODE) users[u.userId].logRecommendation(rec);
        int newContentToWatch = rec.getRandomRecommendation();

        Event newEvent = new Event(generator.nextInt(30), "User", u.userId,
             Event.eventType.USER_STARTS_WATCHING, newContentToWatch);
        eventQueue.add(newEvent);
      }
    }

    private static void InitializeSystem(int usersPerSupernode, String[]
       topicarray, int numberOfVideos, int numberOfVideosPerSupernode,
       superNodeArchitecture architecture) {
      Random generator = new Random();
      topics = new Topic[topicarray.length];

      // Generate topics
      for (int topicId = 0; topicId < topicarray.length; topicId++) {
        Topic newTopic = new Topic(topicId, topicarray[topicId]);
        topics[topicId] = newTopic;
      }

      // Generate content
      for (int contentId = 0; contentId < numberOfVideos; contentId++) {
        Topic relatedTopic = topics[generator.nextInt(topics.length)];
        Topic[] relatedTopics = new Topic[1];
        relatedTopics[0] = relatedTopic;
        Content newContent = new Content(
            contentId, //
            generator.nextInt(100),       // BaseRank is in [0, 99] -> 99
                highest rank
            relatedTopics,                // Currently only relevant for one
                 topic
            generator.nextInt(291) + 10   // Assume length is in [10, 300]
          );
        contents.add(newContent);
```

```
        System.out.println("Generated new content: " + contents.get(
            contentId).getContentId() + ", " + contents.get(contentId).
            getBaseRank() + ", " + contents.get(contentId).
            getRelevantTopics()[0].getTopicName() + ", " + contents.get(
            contentId).getLength());
      }

      // Generate supernodes and users. And add content to supernodes
      users = new User[supernodes.length * usersPerSupernode];
      System.out.println("Initializing " + supernodes.length + " supernodes
          , each with " + usersPerSupernode + " users...");
      for (int supernodeId = 0; supernodeId < supernodes.length;
          supernodeId++) {
        System.out.println("Generating Supernode " + supernodeId + "...");
        supernodes[supernodeId] = new Supernode(supernodeId);
        for (int userId = 0; userId < usersPerSupernode; userId++) {
          User newUser = new User(userId + supernodeId*usersPerSupernode,
              supernodeId, topics[generator.nextInt(topics.length)]);
          System.out.println("  ...generated user with id " + newUser.
              getUserId() + " and interest " + newUser.getTopicInterest().
              topicName +
              " (belonging to supernode " + newUser.getSupernodeServer() +
                  ").");
          users[userId + supernodeId*usersPerSupernode] = newUser;
          supernodes[supernodeId].addUser(newUser);
        }
        for (int contentId = 0; contentId < numberOfVideosPerSupernode;
            contentId++) {
          supernodes[supernodeId].addContent(contents.get(generator.nextInt
              (contents.size())), 0);
        }
      }

      // Set up Supernode neighborhood.
      InitializeNeighborhood(architecture);
    }

    private static void InitializeNeighborhood(superNodeArchitecture arch)
        {
      switch (arch) {
      case RING:
        for (Supernode s: supernodes) {
          s.addNeighbor(supernodes[(s.supernodeId+1) % supernodes.length]);
        }
        return;
      case DOUBLERING:
        for (Supernode s: supernodes) {
          s.addNeighbor(supernodes[(s.supernodeId+1) % supernodes.length]);
          s.addNeighbor(supernodes[(s.supernodeId+supernodes.length-1) %
              supernodes.length]);
        }
        return;
      case TWORINGS:
        for (Supernode s: supernodes) {
          s.addNeighbor(supernodes[(s.supernodeId+1) % supernodes.length]);
          s.addNeighbor(supernodes[(s.supernodeId+(supernodes.length / 2))
              % supernodes.length]);
        }
      case ALLCONNECTED:
        for (Supernode s: supernodes) {
          for (int i = 0; i < supernodes.length; i++) {
            if (s.supernodeId != i) {  // Should not send gossip to oneself
                ...
              s.addNeighbor(supernodes[i]);
            }
          }
```

```
        }
        return;
350   }
      return;
    }

    private static void logCumulativeBandwidth(int time) {
      ArrayList<CumulativeBandwidth> bws = new ArrayList<
          CumulativeBandwidth>();
      for (Content c: contents) {
        bws.add(new CumulativeBandwidth(c.contentId, c.cumulativeBandwidth)
            );
      }
      Comparator<CumulativeBandwidth> comp = new CumulativeBandwidth.Comp()
          ;
360   Collections.sort(bws, comp);
      try {
        FileWriter log = new FileWriter(logDirectory + "bandwidths.log",
            true);
        log.write("Time " + String.format("%4d", time) + ": Here are the
            cumulative bandwidths:\n");
        for (int i = 0; i < bws.size(); i++) {
          log.write("Content " + String.format("%4d", bws.get(i).contentId)
              + ": " + String.format("%3d", bws.get(i).bandwidth) + "\n");
        }
        log.close();
      } catch (IOException ex) {
        System.out.println("Whoops: Could not log cumulative bandwidths: "
            + ex.getMessage());
370   }
      bws.clear();
      return;
    }

    private static void logSuperContentCreation(int time, Content
        newContent, int supernodeId) {
      try {
        FileWriter log = new FileWriter(logFile, true);
        log.write("Time " + String.format("%4d", time) + ": Supernode " +
            String.format("%2d", supernodeId) + ": Super content with id "
            + newContent.contentId + " generated.\n");
        log.close();
380   } catch (IOException ex) {
        System.out.println("Whops: Could not log super content creation: "
            + ex.getMessage());
      }

    }

    private static void BuildIndex(Supernode s) {
      System.out.print("Building index for supernode " + s.getSupernodeId()
          + "...");
      for (Content c: s.getServesContent()) {
        IndexElement ie = new IndexElement(c.contentId, c.baseRank, 0);
390     s.addIndexElementToIndex(c.relevantTopics[0], ie);
      }
      System.out.println("Done!");
      s.logEntireIndex();
    }

    private static void DebugPrintSizes() {
      System.out.println("Here are the sizes of the system:");
      System.out.println("Supernodes: " + supernodes.length);
      System.out.println("Users     : " + users.length);
400   System.out.println("Contents  : " + contents.size());
      System.out.println("Topics    : " + topics.length);
```

```
    System.out.println("Contents per supernode:");
    for (int i=0; i < supernodes.length; i++) {
      System.out.println("  " + i + ": " + supernodes[i].getServesContent
          ().size());
    }
  }
}
```

A.1.2 Supernode.java

```
package master;

import master.User;
import master.Content;
import master.Recommendation;
import master.Gossip;
import master.IndexEntry;
import master.IndexElement;

import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

/**
 * Supernode.java
 * Purpose: This is the class representing all supernodes in the system.
 *          It takes care of content, a local search index, and lists of
 *          connected users and neighbor supernodes.
 *
 * @author Torkil Grindstein
 * @version 1.0
 *
 */
public class Supernode {
  int supernodeId;
  ArrayList<User> servingUsers = new ArrayList<User>();
  ArrayList<Content> servesContent = new ArrayList<Content>();
  ArrayList<IndexEntry> index = new ArrayList<IndexEntry>();
  ArrayList<Supernode> neighbors = new ArrayList<Supernode>();
  ArrayList<Gossip> incomingGossip = new ArrayList<Gossip>();
  boolean needsRecalculation = false;

  String logDirectory = "/home/torkil/workspace/Master/log/supernodes/";
  String logFile;
  String indexFile;
  String measureFile;

  enum sortBy {RANK, FORM, TIME};

  public Supernode(int supernodeId) {
    this.supernodeId = supernodeId;
    this.logFile = logDirectory + "Supernode-" + supernodeId + ".log";
    this.indexFile = logDirectory + "Supernode-" + supernodeId + "-index.
        log";
    this.measureFile = logDirectory + "../measures/Simulation.log";

    logSupernodeCreation();
  }

  public int getSupernodeId() {
    return supernodeId;
  }
```

```
  public void setSupernodeId(int supernodeId) {
    this.supernodeId = supernodeId;
  }

  public ArrayList<User> getServingUsers() {
    return servingUsers;
  }

  public void setServingUsers(ArrayList<User> servingUsers) {
    this.servingUsers = servingUsers;
  }

  public ArrayList<Content> getServesContent() {
    return servesContent;
  }

  public void setServesContent(ArrayList<Content> servesContent) {
    this.servesContent = servesContent;
  }

  public ArrayList<IndexEntry> getIndex() {
    return index;
  }

  public void setIndex(ArrayList<IndexEntry> index) {
    this.index = index;
  }

  public boolean isNeedsRecalculation() {
    return needsRecalculation;
  }

  public void setNeedsRecalculation(boolean needsRecalculation) {
    this.needsRecalculation = needsRecalculation;
  }

  public ArrayList<Supernode> getNeighbors() {
    return neighbors;
  }

  public void setNeighbors(ArrayList<Supernode> neighbors) {
    this.neighbors = neighbors;
  }

  public void addUser(User newUser) {
    this.servingUsers.add(newUser);

    logUserAdd(newUser.getUserId());
  }

  public void addNeighbor(Supernode newNeighbor) {
    this.neighbors.add(newNeighbor);

    logNeighborAdd(newNeighbor.getSupernodeId());
  }

  public void addContent(Content newContent, int time) {
    // Verify that the content is not already present
    boolean contentAlreadyPresent = false;
    for (Content presentContent: servesContent) {
      if (presentContent.contentId == newContent.contentId) {
        contentAlreadyPresent = true;
        continue;
      }
    }
    if (!contentAlreadyPresent) {
```

```
        this.servesContent.add(newContent);
        logContentAdd(time, newContent);
        newContent.cumulativeBandwidth++;
        newContent.logServedBy(this.supernodeId, time);
      }
    }

    public void removeContent(Content c) {
      this.servesContent.remove(c);
130   }

    public void addIndexElementToIndex(Topic t, IndexElement ie) {
      // Check if topic is present
      for (IndexEntry indexEntry: index) {
        if (indexEntry.entry.containsKey(t)) { // Yes, and just add another
            element
          indexEntry.entry.get(t).add(ie);
          return;
        }
      }
140   // Could not find topic, hence a new entry is required
      IndexEntry newEntry = new IndexEntry(t, ie);
      index.add(newEntry);
    }

    public void removeFromIndex(Topic t, IndexElement ie) {
      // Find correct topic
      for (IndexEntry indexEntry: index) {
        if (indexEntry.entry.containsKey(t)) { // Yes
          indexEntry.entry.get(t).remove(ie);
150       logRemovalFromIndex(t, ie);
          return;
        }
      }
    }

    private int recommendByField(ArrayList<IndexElement> elements,
        ArrayList<Integer> haveSeen) {
      for (IndexElement e: elements) {
        if (e.contentId >= 0) {
          if (haveSeen.size() > 0) {
160         boolean thisOneIsNew = true;
            for (int i=0; i < haveSeen.size(); i++) {
              if (e.contentId == haveSeen.get(i)) {  // Video already seen
                  ...
                thisOneIsNew = false;
              }
            }
            if (thisOneIsNew) {
              return e.contentId;
            }
          } else {
170         return e.contentId;
          }
        }
      }
      // Sorry, couldn't recommend anything to you...
      return -1;
    }

    public Recommendation recommend(Topic t, ArrayList<Integer> haveSeen) {
      Recommendation rec = new Recommendation(-1, -1, -1);
180   for (IndexEntry ie: index) {
        if (ie.entry.containsKey(t)) {
          sortIndex(ie.entry.get(t), sortBy.RANK);
          rec.rankRecommendation = recommendByField(ie.entry.get(t),
```

```
            haveSeen);
        sortIndex(ie.entry.get(t), sortBy.FORM);
        rec.upAndComingRecommendation = recommendByField(ie.entry.get(t),
             haveSeen);
        sortIndex(ie.entry.get(t), sortBy.TIME);
        rec.newcomerRecommendation = recommendByField(ie.entry.get(t),
            haveSeen);
      }
    }
    return rec;
  }

  public static class Comp implements Comparator<IndexElement> {
    private sortBy sortOn;

    public Comp(sortBy sortOn) {
      this.sortOn = sortOn;
    }

    @Override
    public int compare(IndexElement o1, IndexElement o2) {
      switch(sortOn) {
      case RANK:
        return new Integer(o2.smartRank).compareTo(o1.smartRank);
      case FORM:
        return new Integer(o2.numberOfPositives - o2.numberOfNegatives).
            compareTo(o1.numberOfPositives - o1.numberOfNegatives);
      case TIME:
        return new Integer(o2.timeAdded).compareTo(o1.timeAdded);
      }
      return -1;
    }
  }

  public void spreadGossip(int time, Content c, Topic t) {
    Gossip g = new Gossip(c, t);
    for (Supernode neighbor: this.neighbors) {
      neighbor.incomingGossip.add(g);
      logSpreadGossip(time, neighbor, g);
    }
    return;
  }

  public void receiveGossip(int time) {
    Gossip g = new Gossip();
    while (this.incomingGossip.size() > 0) {
      // Check if this Supernode already serves the gossipped content
      g = this.incomingGossip.get(0);
      logReceivedGossip(time, g);
      if (!this.servesContent.contains(g.content)) {
        this.addContent(g.content, time);
        // Add to index
        IndexElement ie = new IndexElement(g.content.contentId, g.content
            .baseRank, time);
        this.addIndexElementToIndex(g.content.relevantTopics[0], ie);
      }
      this.incomingGossip.remove(0);
    }
    return;
  }

  private void sortIndex(ArrayList<IndexElement> unsorted, sortBy sortOn)
      {
    Comparator<IndexElement> comp = new Comp(sortOn);
    Collections.sort(unsorted, comp);
  }
```

```
    private void logSupernodeCreation() {
      try {
        FileWriter log = new FileWriter(logFile, true);
        log.write("
            ##########################################################################################################################
            n");
        log.write("#\n");
        log.write("# Supernode " + supernodeId + " successfully created.\n"
            );
        log.write("#\n");
        log.close();
      } catch (IOException ex) {
        System.out.println("Whops: Could not log supernode creation: " + ex
            .getMessage());
      }
    }

    private void logUserAdd(int userId) {
      try {
        FileWriter log = new FileWriter(logFile, true);
        log.write("User " + userId + " is now served.\n");
        log.close();
      } catch (IOException ex) {
        System.out.println("Whops: Could not log adding of user: " + ex.
            getMessage());
      }
    }

    private void logNeighborAdd(int supernodeId) {
      try {
        FileWriter log = new FileWriter(logFile, true);
        log.write("I am now a neighbor with Supernode " + supernodeId + ".\
            n");
        log.close();
      } catch (IOException ex) {
        System.out.println("Whops: Could not log adding of neighbor: " + ex
            .getMessage());
      }
    }

    private void logSpreadGossip(int time, Supernode neighbor, Gossip g) {
      try {
        String logString = "I spread gossip about " + g.content.contentId +
             " for topic " + g.topic.topicName + " to Supernode " +
            neighbor.supernodeId + ".\n";
        FileWriter log = new FileWriter(logFile, true);
        log.write(logString);
        log.close();
//        if (g.content.isSuperContent) {
//          FileWriter measure = new FileWriter(measureFile, true);
//          measure.write(this.supernodeId + " (time = " + time + "): " +
    logString);
//          measure.close();
//        }
      } catch (IOException ex) {
        System.out.println("Whops: Could not log spreading of gossip: " +
            ex.getMessage());
      }
    }

    private void logReceivedGossip(int time, Gossip g) {
      try {
        String logString = "I received gossip about " + g.content.contentId
             + " for topic " + g.topic.topicName + ".\n";
        FileWriter log = new FileWriter(logFile, true);
```

```
        log.write(logString);
        log.close();
//        if (g.content.isSuperContent) {
//          FileWriter measure = new FileWriter(measureFile, true);
//          measure.write(this.supernodeId + " (time = " + time + "): " +
    logString);
//          measure.close();
//        }
      } catch (IOException ex) {
        System.out.println("Whops: Could not log receival of gossip: " + ex
            .getMessage());
      }
    }

    private void logContentAdd(int time, Content c) {
      try {
        String logString = "Content " + c.contentId + " is now stored.\n";
        FileWriter log = new FileWriter(logFile, true);
        log.write(logString);
        log.close();
        if (c.isSuperContent) {
          FileWriter measure = new FileWriter(measureFile, true);
          measure.write("Time " + String.format("%4d", time) + ": Supernode
               " + String.format("%2d", this.supernodeId) + ": Super
              content with id " + c.contentId + " added.\n");
          measure.close();
        }
      } catch (IOException ex) {
        System.out.println("Whops: Could not log adding of content: " + ex.
            getMessage());
      }
    }

    private void logRemovalFromIndex(Topic t, IndexElement ie) {
      try {
        FileWriter log = new FileWriter(logFile, true);
        log.write("Content " + ie.contentId + " is now removed from my
            index (under topic " + t.topicName + ").\n");
        log.close();
      } catch (IOException ex) {
        System.out.println("Whops: Could not log removal from inde: " + ex.
            getMessage());
      }
    }

    public void logEntireIndex() {
      try {
        FileWriter log = new FileWriter(indexFile, true);
        log.write("\nHere is the entire index for Supernode " + supernodeId
             + ":\n");
        for (IndexEntry topic: index) {
          for (Topic t: topic.entry.keySet()) {
            log.write("  Topic " + t.topicName + "\n");
            sortIndex(topic.entry.get(t), sortBy.RANK);
            for (IndexElement content: topic.entry.get(t)) {
              if (content.contentId >= 0) {
                log.write("    Id " + String.format("%3d", content.
                    contentId) +
                    ": SmartRank " + String.format("%3d", content.smartRank
                        ) +
                    ", BaseRank: " + String.format("%3d", content.baseRank)
                         +
                    ", Positives: " + String.format("%2d", content.
                        numberOfPositives) +
                    ", Negatives: " + String.format("%2d", content.
                        numberOfNegatives) +
```

```
                    ", Added: " + String.format("%4d", content.timeAdded) +
                    ".\n");
                }
              }
            }
          }
          log.close();
        } catch (IOException ex) {
          System.out.println("Whops: Could not log the index: " + ex.
              getMessage());
        }
      }
    }
```

A.1.3 User.java

```
package master;

import master.Content;
import java.util.ArrayList;
import java.io.FileWriter;
import java.io.IOException;

/**
 * User.java
 * Purpose:
 *
 * @author Torkil Grindstein
 * @version 1.0
 *
 */
public class User {
  int userId;
  int supernodeServer = -1;
  Topic topicInterest;
  Content currentlyWatching = null;
  int timeStartedWatching;
  ArrayList<Integer> haveSeen = new ArrayList<Integer>();

  String logDirectory = "/home/torkil/workspace/Master/log/users/";
  String logFile;

  /**
   * Constructor. It sets userId and supernodeServer
   *
   * @param userId      The user id of the newly created user
   * @param supernodeServer The id for the supernode to which requests
       will be sent
   */
  public User(int userId, int supernodeServer) {
    this.userId = userId;
    this.supernodeServer = supernodeServer;
    this.logFile = logDirectory + "User-" + userId + ".log";

    logUserCreation();
  }

  /**
   * Constructor. It sets userId, supernodeServer and topicInterest
   *
   * @param userId      The user id of the newly created user
   * @param supernodeServer The id for the supernode to which requests
       will be sent
   * @param topicInterest Some topic the user will search for
   */
```

```
  public User(int userId, int supernodeServer, Topic topicInterest) {
    this.userId = userId;
    this.supernodeServer = supernodeServer;
    this.topicInterest = topicInterest;
    this.logFile = logDirectory + "User-" + userId + ".log";

    logUserCreation();
  }

  /**
   * Getter for userId
   *
   * @return  This users user id
   */
  public int getUserId() {
    return userId;
  }

  /**
   * Setter for userId
   *
   * @param userId  The user id to be set
   */
  public void setUserId(int userId) {
    this.userId = userId;
  }

  /**
   * Getter for topicInterest
   *
   * @return  The user's topic of interest
   */
  public Topic getTopicInterest() {
    return topicInterest;
  }

  /**
   * Setter for topicInterest
   *
   * @param topicInterest The user's topic of interest
   */
  public void setTopicInterest(Topic topicInterest) {
    this.topicInterest = topicInterest;
  }

  /**
   * Getter for supernodeServer
   *
   * @return  The user's connected supernode server id
   */
  public int getSupernodeServer() {
    return supernodeServer;
  }

  /**
   * setter for supernodeServer
   *
   * @param supernodeServer The user's connected supernode server id
   */
  public void setSupernodeServer(int supernodeServer) {
    this.supernodeServer = supernodeServer;
    logSetSupernode();
  }

  /**
   * Getter for getCurrentlyWatching
```

```
     *
     * @return  The id of the multimedia content currently being watched by
         the user
     */
    public Content getCurrentlyWatching() {
      return currentlyWatching;
    }

    /**
     * Setter for getCurrentlyWatching
     *
     * @param currentlyWatching The id of the multimedia content currently
         being watched by the user
     */
    public void setCurrentlyWatching(Content currentlyWatching) {
      this.currentlyWatching = currentlyWatching;
    }

    /**
     * Getter for timeStartedWatching
     *
     * @return  The timestamp when the user started to watch currently
         watched video
     */
    public int getTimeStartedWatching() {
      return timeStartedWatching;
    }

    /**
     * Setter for timeStartedWatching
     *
     * @param timeStartedWatching The timestamp when the user started to
         watch currently watched video
     */
    public void setTimeStartedWatching(int timeStartedWatching) {
      this.timeStartedWatching = timeStartedWatching;
    }

    /**
     * Checking whether the user is active, meaning currently watching, or
         not
     *
     * @return  boolean value telling whether the user is watching or not
     */
    public boolean isActive() {
      if (currentlyWatching == null) {
        return false;
      } else {
        return true;
      }
    }

    /**
     * Method to update list of watched content
     *
     * @param c The content id of the recently watched video
     */
    public void justSawContent(Content c) {
      haveSeen.add(c.contentId);
    }

    /**
     * Method that logs user creation in a human readable manner
     * No parameters or return values
     */
    private void logUserCreation() {
```

```
    String supernodeString;
    if (supernodeServer != -1) {
      supernodeString = " (served by Supernode " + Integer.toString(
          supernodeServer) + ").";
    } else {
      supernodeString = ".";
    }
    String topic;
    if (topicInterest != null) {
      topic = " with interest \"" + topicInterest.topicName + "\"";
    } else {
      topic = "";
    }
    try {
      FileWriter log = new FileWriter(logFile, true);
      log.write("\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
      log.write("
          ##############################################################################################################
          n");
      log.write("#\n");
      log.write("# User " + userId + topic + " successfully created" +
          supernodeString + "\n");
      log.write("#\n\n");
      log.close();
    } catch (IOException ex) {
      System.out.println("Whops: Could not log user creation: " + ex.
          getMessage());
    }
  }

  /**
   * Method that logs the connection between the user and a supernode
   * No parameters or return values
   */
  private void logSetSupernode() {
    try {
      FileWriter log = new FileWriter(logFile, true);
      log.write("SupernodeServer is now set to " + supernodeServer + ".\n
          ");
      log.close();
    } catch (IOException ex) {
      System.out.println("Whops: Could not log setting of supernode
          server: " + ex.getMessage());
    }
  }

  /**
   * Method that logs which three recommendations it received from its
       supernode
   * No parameters or return values
   *
   * @param rec The recommendation object containing all three
       recommendatations
   */
  public void logRecommendation(Recommendation rec) {
    try {
      FileWriter log = new FileWriter(logFile, true);
      log.write("My Supernode recommends (for topic " + this.
          topicInterest.topicName + "): ");
      log.write("Rank: " + rec.rankRecommentation + ", form: " + rec.
          upAndComingRecommendation + ", time: " + rec.
          newcomerRecommendation + ".\n");
      log.close();
    } catch (IOException ex) {
      System.out.println("Whops: Could not log recommendation: " + ex.
          getMessage());
```

```
    }
  }

  /**
   * Method that logs when a user starts to watch a video
   *
   * @param time     The timestamp when the user starts watching
   * @param contentid The video's id
   * @param length     The length of the video
   */
  public void logStartsWatching(int time, int contentid, int length) {
    try {
      FileWriter log = new FileWriter(logFile, true);
      log.write("At time " + time + " I start watching content " +
          contentid + " (with length " + length + ").\n");
      log.close();
    } catch (IOException ex) {
      System.out.println("Whops: Could not log start watching: " + ex.
          getMessage());
    }
  }

  /**
   * Method that logs that a user is still watching a specific piece of
       content.
   * This method is only called when the time simulation ticks once per
       time unit.
   * Usually, the other time simulation method is applied.
   *
   * @param time     The current timestamp
   * @param contentid The video's id
   * @param timeLeft  The number of time units until the video is
       finished
   */
  public void logStillWatching(int time, int contentid, int timeLeft) {
    try {
      FileWriter log = new FileWriter(logFile, true);
      String tl = ((Integer) timeLeft).toString();
      if ((timeLeft % 100) == 0) {
        log.write("C");
      } else if ((timeLeft % 10) == 0) {
        log.write("X");
      } else if (timeLeft < 10) {
        log.write(tl);
      } else {
        log.write(".");
      }
      log.close();
    } catch (IOException ex) {
      System.out.println("Whops: Could not log finished watching: " + ex.
          getMessage());
    }
  }

  /**
   * Method that logs when a user is finished watching a specific video
   *
   * @param time     The current timestamp
   * @param contentid The video's id
   * @param rate     The user's evaluation of the video - either "good" or
       "bad"
   */
  public void logFinishedWatching(int time, int contentid, String rate) {
    try {
      FileWriter log = new FileWriter(logFile, true);
      log.write("\nAt time " + time + " I finished watching content " +
```

```
            contentid + " and rated it " + rate + "\n");
        log.close();
      } catch (IOException ex) {
        System.out.println("Whops: Could not log finished watching: " + ex.
            getMessage());
290   }
    }

    /**
     * Method that logs that the supernode is out of recommendations for
         the users
     * topic of interest. In the cases where this happens, the users stays
         idle for some
     * time, and checks with the supernode later if something new has
         arrived.
     *
     * @param time  The current timestamp
     */
300 public void logNothingAvailable(int time) {
      try {
        FileWriter log = new FileWriter(logFile, true);
        log.write("At time " + time + " my Supernode could not recommend
            anything...\n");
        log.close();
      } catch (IOException ex) {
        System.out.println("Whops: Could not log no availability: " + ex.
            getMessage());
      }
    }
}
```

A.1.4 Topic.java

```
package master;

/**
 * Topic.java
 * Purpose: This class contains needed information per topic (search
     queries)
 *
 * @author Torkil Grindstein
 * @version 1.0
 *
10 */
public class Topic {
  int topicId;
  String topicName;

  /**
   * Constructor setting an id and topic name
   *
   * @param topicId The id for this topic
   * @param topicName The topic (search query) name
20 */
  public Topic(int topicId, String topicName) {
    this.topicId = topicId;
    this.topicName = topicName;
  }

  /**
   * Getter for topicId
   *
   * @return The id for this topic
30 */
  public int getTopicId() {
```

```
    return topicId;
  }

  /**
   * Setter for topicId
   *
   * @param topicId The id for this topic
   */
  public void setTopicId(int topicId) {
    this.topicId = topicId;
  }

  /**
   * Getter for the topicName
   *
   * @return  A string containing the name of the topic
   */
  public String getTopicName() {
    return topicName;
  }

  /**
   * Setter for the topicName
   *
   * @param topicName A string containing the name of the topic
   */
  public void setTopicName(String topicName) {
    this.topicName = topicName;
  }
}
```

A.1.5 Content.java

```
package master;

import java.util.Random;
import java.io.FileWriter;
import java.io.IOException;

import master.Topic;

/**
 * Content.java
 * Purpose: This class stores multimedia content
 *          Among attributes are ranking, length and for which search
 *          queries it is interesting. It also contains a hidden
    attribute
 *          for how popular it _really_ is among users.
 *
 * @author Torkil Grindstein
 * @version 1.0
 *
 */
public class Content {
  int contentId;
  int baseRank;
  Topic[] relevantTopics;
  int length;
  double realPopularity;
  boolean isSuperContent = false;
  int cumulativeBandwidth;

  String logDirectory = "/home/torkil/workspace/Master/log/contents/";
  String logFile;
  Random generator = new Random();
```

```
    /**
     * Constructor. It sets contentId, baseRank, relevantTopics and length
     *
     * @param contentId   The content id for this video
     * @param baseRank    The base (original) ranking value for this video
     * @param relevantTopics  The topics for which this content shall
         produce search results
     * @param length      The video's length in terms of time units
     */
    public Content(int contentId, int baseRank, Topic[] relevantTopics, int
         length) {
      this.contentId = contentId;
      this.baseRank = baseRank;
      this.relevantTopics = relevantTopics;
      this.length = length;
      this.realPopularity = generator.nextDouble();
      this.cumulativeBandwidth = 0;
      this.logFile = logDirectory + "Content-" + contentId + ".log";

      logContentsCreation();
    }

    /**
     * Constructor. It sets contentId, baseRank, relevantTopics, length,
         popularity and more
     *
     * @param contentId   The content id for this video
     * @param baseRank    The base (original) ranking value for this video
     * @param relevantTopics  The topics for which this content shall
         produce search results
     * @param length      The video's length in terms of time units
     * @param popularity    The video's real popularity. Used as hidden
         value and important in evaluation
     * @paran isSuper     Boolean value telling if this is so called "super
          content", which everybody likes.
     *              Used for testing and evaluation.
     */
    public Content(int contentId, int baseRank, Topic[] relevantTopics, int
         length, double popularity, boolean isSuper) {
      this.contentId = contentId;
      this.baseRank = baseRank;
      this.relevantTopics = relevantTopics;
      this.length = length;
      this.realPopularity = popularity;
      this.cumulativeBandwidth = 0;
      this.isSuperContent = isSuper;
      this.logFile = logDirectory + "Content-" + contentId + ".log";

      logContentsCreation();
    }

    /**
     * Getter for contentId
     *
     * @return  The content's id
     */
    public int getContentId() {
      return contentId;
    }

    /**
     * Setter for contentId
     *
     * @param contentid The content's id
     */
```

```
  public void setContentId(int contentId) {
    this.contentId = contentId;
  }

  /**
   * Getter for baseRank
   *
   * @return  The video's base rank
   */
  public int getBaseRank() {
    return baseRank;
  }

  /**
   * Setter for baseRank
   *
   * @param baseRank  The video's base rank
   */
  public void setBaseRank(int baseRank) {
    this.baseRank = baseRank;
  }

  /**
   * Getter for relevantTopics
   *
   * @return  The video's relevant topic list
   */
  public Topic[] getRelevantTopics() {
    return relevantTopics;
  }

  /**
   * Setter for relevantTopics
   *
   * @param relevantTopics  The video's relevant topic list
   */
  public void setRelevantTopics(Topic[] relevantTopics) {
    this.relevantTopics = relevantTopics;
  }

  /**
   * Getter for length
   *
   * @return  The video's length in time units
   */
  public int getLength() {
    return length;
  }

  /**
   * Setter for length
   *
   * @param length  The video's length in time units
   */
  public void setLength(int length) {
    this.length = length;
  }

  /**
   * Getter for realPopularity
   *
   * @return  The video's real popularity among users
   */
  public double getRealPopularity() {
    return realPopularity;
  }
```

```
    /**
     * Setter for realPopularity
     *
     * @param realPopularity  The video's real popularity among users
     */
    public void setRealPopularity(double realPopularity) {
      this.realPopularity = realPopularity;
    }

    /**
     * Boolean test on whether the video is "super content" (what everybody
         likes)
     *
     * @return  The boolean value indicating whether this is "super content
         " or not
     */
    public boolean isSuperContent() {
      return isSuperContent;
    }

    /**
     * Setter for isSuperContent
     *
     * @param isSuperContent  The boolean value telling whether this is "
         super content" or not
     */
    public void setSuperContent(boolean isSuperContent) {
      this.isSuperContent = isSuperContent;
    }

    /**
     * Getter for cumulativeBandwidth
     *
     * @return  The cumulative bandwidth for this video. This is a function
         of the number of
     *          supernodes who host this video
     */
    public int getCumulativeBandwidth() {
      return cumulativeBandwidth;
    }

    /**
     * Setter for cumulativeBandwidth
     *
     * @param cumulativeBandwidth The cumulative bandwidth for this video.
         This is a function of the number of
     *                supernodes who host this video
     */
    public void setCumulativeBandwidth(int cumulativeBandwidth) {
      this.cumulativeBandwidth = cumulativeBandwidth;
    }

    /**
     * Method that logs content creation in a human readable manner
     * No parameters or return values
     */
    public void logContentsCreation() {
      try {
        FileWriter log = new FileWriter(logFile, true);
        log.write("#################### Created ####################\n");
        log.close();
      } catch (IOException ex) {
        System.out.println("Whops: Could not log content creation: " + ex.
            getMessage());
      }
```

```
    }

    /**
220  * Method that logs which supernode that serves this content
     *
     * @param supernodeId The id of the hosting supernode
     * @param time    The timestamp for when the supernode added this video
     */
    public void logServedBy(int supernodeId, int time) {
      try {
        FileWriter log = new FileWriter(logFile, true);
        String lmsg = String.format("At time %4d I am served by Supernode
            %2d (%2d)\n" +
            "", time, supernodeId, this.cumulativeBandwidth);
230     log.write(lmsg);
        log.close();
      } catch (IOException ex) {
        System.out.println("Whops: Could not log content served by: " + ex.
            getMessage());
      }
    }
  }
```

A.1.6 Recommendation.java

```
  package master;

  import java.util.Random;

  /**
   * Recommendation.java
   * Purpose: This class stores recommendations of all three kinds
   *
   * @author Torkil Grindstein
10 * @version 1.0
   *
   */
  public class Recommendation {
    int rankRecommentation;
    int upAndComingRecommendation;
    int newcomerRecommendation;

    Random generator;

20  /**
     * Constructor, setting three recommended contents (each along each
         axis)
     * The caller of the constructor has a way of nowing for which content
         these recommendations apply.
     *
     * @param rankRecommentation     The content id for the recommendation
         based on smartRank
     * @param upAndcomingRecommendation The content id for the
         recommendation based on up-and-coming videos
     * @param newcomerRecommendation  The content id for the recommendation
          based on brand new material
     */
    public Recommendation(int rankRecommentation, int
        upAndcomingRecommendation, int newcomerRecommendation) {
      this.rankRecommentation = rankRecommentation;
30    this.upAndComingRecommendation = upAndcomingRecommendation;
      this.newcomerRecommendation = newcomerRecommendation;
      this.generator = new Random();

    }
```

```
    /**
     * Getter for the rankRecommendation
     *
     * @return  The content id for the recommendation based on smartRank
     */
    public int getRankRecommentation() {
      return rankRecommentation;
    }

    /**
     * Setter for the rankRecommendation
     *
     * @param rankRecommentation The content id for the recommendation
         based on smartRank
     */
    public void setRankRecommentation(int rankRecommentation) {
      this.rankRecommentation = rankRecommentation;
    }

    /**
     * Getter for the upAndComingRecommendation
     *
     * @return The content id for the recommendation based on recent
         popularity
     */
    public int getUpAndcomingRecommendation() {
      return upAndComingRecommendation;
    }

    /**
     * Setter for the upAndComingRecommendation
     *
     * @param upAndcomingRecommendation The content id for the
         recommendation based on recent popularity
     */
    public void setUpAndcomingRecommendation(int upAndcomingRecommendation)
         {
      this.upAndComingRecommendation = upAndcomingRecommendation;
    }

    /**
     * Getter for the newcomerRecommendation
     *
     * @return The content id for the recommendation based on freshness
     */
    public int getNewcomerRecommendation() {
      return newcomerRecommendation;
    }

    /**
     * Setter for the newcomerRecommendation
     *
     * @param newcomerRecommendation The content id for the recommendation
         based on freshness
     */
    public void setNewcomerRecommendation(int newcomerRecommendation) {
      this.newcomerRecommendation = newcomerRecommendation;
    }

    /**
     * Method that returns the content id of one randomly chosen
         recommendation axis
     *
     * @return The content id for a recommendation, randomly chosen among
         the three axis
```

```
     */
    public int getRandomRecommendation() {
      int rec = generator.nextInt(3);

      if (rec == 0) return rankRecommentation;
      else if (rec == 1) return upAndComingRecommendation;
      else if (rec == 2) return newcomerRecommendation;
      else return -1;
    }
  }
```

A.1.7 Gossip.java

```
  package master;

  /**
   * Gossip.java
   * Purpose: This class contains gossip, that is combinations of contents
       and topics.
   *
   * @author Torkil Grindstein
   * @version 1.0
   *
   */
  public class Gossip {
    Content content;
    Topic topic;

    /**
     * Null constructor
     *
     */
    public Gossip() {
    }

    /**
     * Constructor setting both content and topic
     *
     * @param c The content to gossip about
     * @param t The topic for which the content is relevant
     */
    public Gossip(Content c, Topic t) {
      this.content = c;
      this.topic = t;
    }

    /**
     * Getter for the content
     *
     * @return  The content of the gossip
     */
    public Content getContent() {
      return content;
    }

    /**
     * Setter for the content
     *
     * @param c The content to spread gossip about
     */
    public void setContent(Content c) {
      this.content = c;
    }

    /**
```

```
     * Getter for the topic
     *
     * @return  The topic for which a specific content is relevant
     */
    public Topic getTopic() {
      return topic;
    }

    /**
     * Setter for the topic
     *
     * @param t The topic for which a specific content is relevant
     */
    public void setTopic(Topic t) {
      this.topic = t;
    }
  }
```

A.1.8 IndexElement.java

```
  package master;

  /**
   * IndexElement.java
   * Purpose: This class stores elements used in the search indexes.
   *          Attributes are ranks, counters and timestamp when added in
       the index.
   *
   * @author Torkil Grindstein
   * @version 1.0
   *
   */
  public class IndexElement {
    int contentId;
    int smartRank;
    int baseRank;
    int numberOfPositives = 0;
    int numberOfNegatives = 0;
    int timeAdded = 0;

    /**
     * Constructor setting contentId, baseRank, smartRank and time
     *
     * @param contentId The content id for the content to be indexed
     * @param baseRank  The base rank for the content to be indedexed.
     * @param time    The timestamp for when the content was added to the
         index
     */
    public IndexElement(int contentId, int baseRank, int time) {
      this.contentId = contentId;
      this.baseRank = baseRank;
      this.smartRank = baseRank;
      this.timeAdded = time;
    }

    /**
     * Getter for the content id
     *
     * @return  The content id for the indexed element
     */
    public int getContentId() {
      return contentId;
    }

    /**
```

```
     * Setter for the content id
     *
     * @param contentId The content id for the indexed element
     */
    public void setContentId(int contentId) {
      this.contentId = contentId;
    }

    /**
     * Getter for the smartRank
     *
     * @return  The smartRank value for the indexed element
     */
    public int getSmartRank() {
      return smartRank;
    }

    /**
     * Setter for the smartRank
     *
     * @param smartRank The smartRank value for the indexed element
     */
    public void setSmartRank(int smartRank) {
      this.smartRank = smartRank;
    }

    /**
     * Getter for the baseRank
     *
     * @return  The baseRank for the indexed element
     */
    public int getBaseRank() {
      return baseRank;
    }

    /**
     * Setter for the baseRank
     *
     * @param baseRank  The baseRank for the indexed element
     */
    public void setBaseRank(int baseRank) {
      this.baseRank = baseRank;
    }

    /**
     * Getter for the numberOfPositives
     *
     * @return  The number of positive feedback given by users for the
         indexed element
     */
    public int getNumberOfPositives() {
      return numberOfPositives;
    }

    /**
     * Setter for the numberOfPositives
     *
     * @param numberOfPositives The number of positive feedback given by
         users for the indexed element
     */
    public void setNumberOfPositives(int numberOfPositives) {
      this.numberOfPositives = numberOfPositives;
    }

    /**
     * Getter for the numberOfNegatives
```

```
     *
     * @return  The number of negative feedback given by users for the
         indexed element
     */
    public int getNumberOfNegatives() {
      return numberOfNegatives;
    }

    /**
     * Setter for the numberOfNegatives
     *
     * @param numberOfNegatives The number of negative feedback given by
         users for the indexed element
     */
    public void setNumberOfNegatives(int numberOfNegatives) {
      this.numberOfNegatives = numberOfNegatives;
    }

    /**
     * Getter for the timeAdded
     *
     * @return  The timestamp for when the content was indexed
     */
    public int getTimeAdded() {
      return timeAdded;
    }

    /**
     * Setter for the timeAdded
     *
     * @param timeAdded The timestamp for when the content was indexed
     */
    public void setTimeAdded(int timeAdded) {
      this.timeAdded = timeAdded;
    }
  }
```

A.1.9 IndexEntry.java

```
  package master;

  import java.util.Map;
  import java.util.HashMap;
  import java.util.ArrayList;

  /**
   * IndexEntry.java
   * Purpose: This class stores entries in the search index.
   *          One entry is a map from a search query (aka topic) to a list
   *          of IndexElement objects.
   *
   * @author Torkil Grindstein
   * @version 1.0
   *
   */
  public class IndexEntry {
    Map<Topic, ArrayList<IndexElement>> entry = new HashMap<Topic,
        ArrayList<IndexElement>>();

    /**
     * Null constructor
     *
     */
    public IndexEntry() {
    }
```

```
    /**
     * Constructor, installing one entry in the index
     *
     * @param entry An index entry, consisting of a mapping from a topic to a list of
     *        IndexElement objects, which basically are content
     */
    public IndexEntry(Map<Topic, ArrayList<IndexElement>> entry) {
      this.entry = entry;
    }

    /**
     * Constructor, just adding one specific index element to a topic
     *
     * @param t   The relevant topic
     * @param ie  The index element to add for this topic
     */
    public IndexEntry(Topic t, IndexElement ie) {
      ArrayList<IndexElement> newList = new ArrayList<IndexElement>();
      newList.add(ie);
      entry.put(t, newList);
    }
  }
```

A.1.10 Event.java

```
  package master;

  /**
   * Event.java
   * Purpose: This class stores event information. Events are queued to be
   *          executed in a chronological order. Numerous event types are defined.
   *
   * @author Torkil Grindstein
   * @version 1.0
   *
   */
  public class Event implements Comparable<Object> {
    int timestamp;
    String actor;
    int actorId;
    eventType eventName;
    int objectId; // Can be contentId, userID, etc, depending on the eventType
    Content content;
    Topic topic;

    enum eventType {
      USER_STARTS_WATCHING,
      USER_STOPS_WATCHING,
      USER_IDLE,
      SUPERNODE_SPREADS_GOSSIP,
      SUPERNODE_RECEIVES_GOSSIP,
      SUPERNODE_FETCHES_VIDEO_FROM_SERVER,
      SUPERNODE_RECALCULATION,
      NEW_CONTENT,
      ADD_SUPER_CONTENT,
      SYSTEM_PROGRESS,
      SYSTEM_TERMINATION
    }

    /**
     * Constructor. It sets timestamp, actor/object info and event type
```

```
     *
     * @param timestamp Defines when an event is scheduled to occur
     * @param actor   The actor (subject) of the event. E.g., "System" or "
         Content"
40   * @param actorId   The id (if exists) for the actor
     * @param eventName The event type. Possible choices are from the list
         above, "eventType"
     * @param objectId  If the event has an object, this is the object's id
     *
     * To understand actor vs object, think of the event that a user starts
         to watch a video.
     * Then the user is the actor, and the video is the object.
     * Mark that not all events have obejcts.
     */
    public Event(int timestamp, String actor, int actorId, eventType
        eventName, int objectId) {
      this.timestamp = timestamp;
50    this.actor = actor;
      this.actorId = actorId;
      this.eventName = eventName;
      this.objectId = objectId;
    }

    /**
     * Constructor. It sets timestamp, actor/object info and event type
     *
     * @param timestamp Defines when an event is scheduled to occur
60   * @param actor   The actor (subject) of the event. E.g., "System" or "
         Content"
     * @param actorId   The id (if exists) for the actor
     * @param eventName The event type. Possible choices are from the list
         above, "eventType"
     * @param content   The content applicable for this event
     * @param topic   The topic applicable for this event
     */
    public Event(int timestamp, String actor, int actorId, eventType
        eventName, Content content, Topic topic) {
      this.timestamp = timestamp;
      this.actor = actor;
      this.actorId = actorId;
70    this.eventName = eventName;
      this.content = content;
      this.topic = topic;
    }

    /**
     * Method to decide the chronological order of two events
     *
     * @param o The other object to compare to this
     * @return  -1  if "this" event occurs before "that"
80   * @return  0 if the events occur simultaneously
     * @return  1 if "that" event occurs before "this"
     */
    public int compareTo(Object o) {
      Event that = (Event) o;
      if (this.timestamp < that.timestamp) return -1;
      if (this.timestamp > that.timestamp) return 1;
      return 0;
    }

90  /**
     * Getter for the timestamp value
     *
     * @return  The timestamp for the event
     */
    public int getTimestamp() {
```

```
    return timestamp;
  }

  /**
   * Setter for the timestamp value
   *
   * @param timestamp The timestamp to set for the event
   */

  public void setTimestamp(int timestamp) {
    this.timestamp = timestamp;
  }

  /**
   * Getter for the actor
   *
   * @return  The actor of the event
   */
  public String getActor() {
    return actor;
  }

  /**
   * Setter for the actor
   *
   * @param actor The actor to set for this event
   */
  public void setActor(String actor) {
    this.actor = actor;
  }

  /**
   * Getter for the actor id
   *
   * @return  This event's actor id
   */
  public int getActorId() {
    return actorId;
  }

  /** Setter for the actor id
   *
   * @param actorId The actor id to set for this event
   */
  public void setActorId(int actorId) {
    this.actorId = actorId;
  }
}
```

A.1.11 CumulativeBandwidth.java

```
package master;

import java.util.Comparator;

/**
 * CumulativeBandwidth.java
 * Purpose: This class is used to measure how much bandwidth specific
     pieces of
 *          content has available.
 *
 * @author Torkil Grindstein
 * @version 1.0
 *
 */
```

```
public class CumulativeBandwidth {
  int contentId;
  int bandwidth;

  /**
   * Constructor. It sets contentId and bandwidth
   *
   * @param contentId The content id for a video
   * @param bandwidth The cumulative bandwidth for the video. Cumulative
       bandwidth
   *            is a function of the number of serving supernodes
   */
  public CumulativeBandwidth(int contentId, int bandwidth) {
    this.contentId = contentId;
    this.bandwidth = bandwidth;
  }

  /**
   * Getter for contentID
   *
   * @return  The video's content id
   */
  public int getContentId() {
    return contentId;
  }

  /**
   * Setter for contentID
   *
   * @param contentId The video's content id
   */
  public void setContentId(int contentId) {
    this.contentId = contentId;
  }

  /**
   * Getter for bandwidth
   *
   * @return  The video's cumulative bandwidth
   */
  public int getBandwidth() {
    return bandwidth;
  }

  /**
   * Setter for bandwidth
   *
   * @param bandwidth The video's cumulative bandwidth
   */
  public void setBandwidth(int bandwidth) {
    this.bandwidth = bandwidth;
  }

  /**
   * Method for comparing two cumulativeBandwidth objects
   *
   * @return  The bandwidth of the video with highest cumulative
       bandwidth
   */
  public static class Comp implements Comparator<CumulativeBandwidth> {

    @Override
    public int compare(CumulativeBandwidth o1, CumulativeBandwidth o2) {
      return new Integer(o2.bandwidth).compareTo(o1.bandwidth);
    }
  }
```

```
}
```