University of Würzburg
Institute of Computer Science
Research Report Series

Towards Efficient Simulation of Large Scale P2P Networks

Tobias Hoßfeld, Andreas Binzenhöfer, Daniel Schlosser[1],
Kolja Eger[2], Jens Oberender, Ivan Dedinski[3], Gerald
Kunzmann[4]

Report No. 371 October 2005

[1] University of Würzburg
Department of Computer Science
Am Hubland, D-97074 Würzburg, Germany
{hossfeld,binzenhoefer,schlosser}@informatik.uni-wuerzburg.de

[2] Hamburg University of Technology (TUHH), Department of Communication Networks
Schwarzenbergstr. 95, D-21073 Hamburg, Germany.
eger@tuhh.de

[3] University of Passau, Chair of Computer Networks and Computer Communications
Innstaße 33, D-94032 Passau, Germany.
{oberender,dedinski}@fmi.uni-passau.de

[4] Technical University of Munich, Institute of Communication Networks
Arcisstr. 21, D-80290 München, Germany.
gerald.kunzmann@tum.de

Presented at: Workshop “P2P Sim”, University of Würzburg, 20.07.2005

Towards Efficient Simulation of Large Scale P2P Networks

Tobias Hoßfeld, Andreas Binzenhöfer, Daniel Schlosser[1], Kolja Eger[2], Jens Oberender, Ivan Dedinski[3], Gerald Kunzmann[4]

[1] University of Würzburg
Department of Computer Science
Am Hubland, D-97074 Würzburg, Germany
{hossfeld,binzenhoefer,schlosser}@informatik.uni-wuerzburg.de

[2] Hamburg University of Technology (TUHH), Department of Communication Networks
Schwarzenbergstr. 95, D-21073 Hamburg, Germany.
eger@tuhh.de

[3] University of Passau, Chair of Computer Networks and Computer Communications
Innstaße 33, D-94032 Passau, Germany.
{oberender,dedinski}@fmi.uni-passau.de

[4] Technical University of Munich, Institute of Communication Networks
Arcisstr. 21, D-80290 München, Germany.
gerald.kunzmann@tum.de

Abstract

The algorithms and methods of the Peer-to-Peer (P2P) technology are often applied to networks and services with a demand for scalability. In contrast to traditional client/server architectures, an arbitrary large number of users, called peers, may participate in the network and use the service without losing any performance. In order to evaluate quantitatively and qualitatively such P2P services and their corresponding networks, different possibilities like analytical apporaches or simulative techniques can be used to improve the implementation of a simulation in general. This task is even more important for large scale P2P networks due to the number of peers, the state space of the P2P network, and the interactions and relationships between peers and states.

The goal of this work is to show how large scale P2P networks can be efficiently evaluated. Methods are demonstrated how to avoid problems occuring in simulations of P2P services. Efficient data structures are required to deal with a large number of events, e.g. the application of a calendar queue for the simulation of a Kademlia-based P2P network, or the priority queue management to simulate eDonkey networks. In order to speed up computational time the simulation has to be implemented in an efficient way, asking for sophisticated programming. This can be achieved for example by using parallel simulation techniques which utilize the distribution and autonomy of the peers in the network.

Appropriate levels of abstraction and models for different application scenarios also improve the computational time for simulations of large scale P2P networks. An example is the simulation of throughput and round trip times in networks. We investigate a BitTorrent network on packet level, thereby, all details on the different layers are taken into account enabling us to study cross-layer interactions.

Next, we take a look on P2P network for signalling in voice/video over IP systems. In this context, the round trip time is crucial for the performance of the system. The packet

layer and its most important characteristics are modeled in order to decrease memory consumption and computational power. Finally, an eDonkey network in a mobile telecommunication system is investigated. In that case we can neglect simulating packets and instead use a stream oriented approach for modeling the transmission of data.

Keywords P2P, eDonkey, Chord, Kademlia, BitTorrent, parallel simulation

1 Introduction

The algorithms and methods of the Peer-to-Peer (P2P) technology are often applied to networks and services with a demand for scalability. In contrast to traditional client/server architectures, an arbitrary large number of users, called peers, may participate in the network and use the service without losing any performance. In order to evaluate quantitatively and qualitatively such P2P services and their corresponding networks, different possibilities like analytical apporaches or simulative techniques can be used to improve the implementation of a simulation in general. This task is even more important for large scale P2P networks due to the number of peers, the state space of the P2P network, and the interactions and relationships between peers and states.

However, the analytical description of the mostly very complex networks leads to problems, as a lot of interactions and parameters have to be considered. A simulation can include all interactions and parameters very easily and may reflect reality as accurately as possible. But the computational power of machines is exceeded very fast. P2P networks with a large number of users also require sufficient memory capacities. It might already be a problem just to keep the states of the peers in the main memory. There are several possibilites to improve the implementation of a simulation in general. This task is even more important for large scale P2P networks. The work is structured as follows.

- Section 2: Efficient Data Structures
  In a P2P network each peer is able to communicate with any other peer. Therefore the number of possible end-to-end connections is in $O(n^2)$ for a network consisting of n peers. This complexity results in a large number of events per time unit. Numerous additional events arise due to the large number of users in a large scale P2P network and specific features of the system like priority queues in eDonkey networks (Section 2.3) or redundancy mechanisms in Chord networks (Sections 2.1). In this section we show how to create solutions to the corresponding problems using efficient datastructures.

- Section 3: Abstractions and Models
  P2P networks can be regarded on different levels of abstractions. Depending on the investigated performance measure and the application scencario it is required to consider the packet level. In other cases, the simulation on application layer is sufficient under given cirumstances. This section deals with the applicability of the different methods. Discussing the advantages and disadvantages of different degrees of abstraction, we show how to find the appropriate level of detail. We present a simulation of BitTorrent on the packet-level using ns-2, Section 3.1. Thereby, all details on the different layers are taken into account enabling us to study cross-layer interactions. Section 3.2 investigates how to model the packet layer and its most important characteristics in order to

signal in voice/video over IP systems using P2P networks. In Section 3.3 we investigate a stream-oriented approach for a P2P file-sharing network which avoids simulating packet for packet. The bandwidth share of eDonkey is used as an example and we approximate the bandwidth that a user gets.

- Section 4: Efficient Programming and Parallel Simulation
  The capability of simulations is mainly restricted by memory and computational power. This difficulty can be overcome with two approaches: efficient implementation, cf. Section 2, or parallel simulation. The parallel simulation benefits from distributed resources, i.e. machines running parts of the simulation, and results in decreasing the running time and increasing the available main memory. This section investigates the applicability of this technique for simulating P2P networks. Problems like synchronisation of logical processes and optimizations regarding the signalling overhead between the machines are considered.

2 Efficient Data Structures

One of the most significant characteristics of a large scale P2P simulation is its enormous complexity. For a network of n peers the number of possible end-to-end connections is already in $O(n^2)$. The huge number of events, interactions and peer states further increases this complexity. Only efficient algorithms and data structures will make fast simulations possible. In this context the running time of the simulation and the required random access memory becomes particularly important. While, to some extent, it is possible to optimize both at the same time, there is usually a trade-off between running time and required memory. The following three factors have the most noticeable influence on this trade-off:

- Efficiency of the event queue
- Internal representation of a state
- Way events are modeled in the simulation

Depending on the investigated problem different kinds of optimization might be preferable. Figure 1 visualizes the arising possibilities. Obviously, the worst case is a completely unoptimized simulation as shown at the bottom of the figure. An efficient implementation of the event queue on the other hand provides an advantage independent of the kind of simulation. In case each peer has to memorize a huge state space, like e.g. the fragmentation of files in eDonkey networks, the optimization of the state representation is especially crucial. If, however, each peer produces a large amount of events, the way events are designed can become the determining factor all of a sudden. In structured P2P networks, e.g., a peer has to maintain events for the stabilization of the overlay, the maintenance of the redundancy, searches and the like. A highly optimized solution as shown on top of Figure 1 incorporates an efficient design of events, a memory saving representation of states and a fast event queue. In the following, we will therefore discuss how to optimize large scale P2P simulations with respect to all three factors. Section 2.1 discusses the advantages and disadvantages of a special priority queue when applied

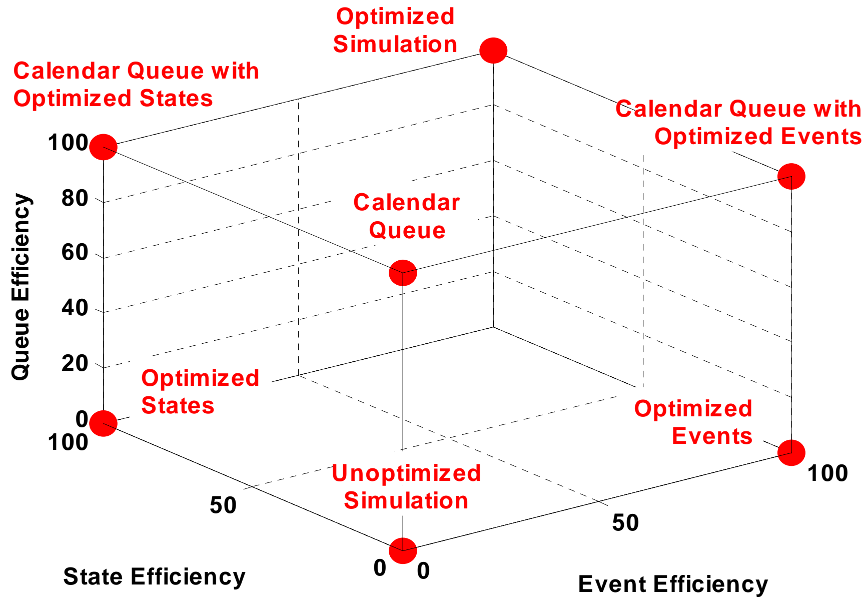


Figure 1: Different categories of simulation efficiency

to the P2P environment. We will present possibilities to adapt the queue to the specific features of a P2P simulation. Considering the waiting queue in eDonkey-like networks as example we will also show how to optimize priority queues with dynamic ranking for special purposes. Since the performance of the queues depends on the number of events and their temporal distribution, we point out the importance of event design algorithms in Section 2.2. Using Kademlia bucket refreshes, we will show how to model periodic and dynamic events efficiently. In Section 2.3 we will introduce two novel approaches to reduce the required memory for the representation of states. The concept of a process handler illustrates how to avoid the redundancy of parallel processes as they frequently occur in large scale P2P systems. As a final example, we will describe how an eDonkey peer can keep track of available and downloaded fragments.

2.1 Priority Queues

A simulative study of the scalability of highly distributed P2P networks automatically involves an immense amount of almost simultaneous events. Due to the large number of peers, a few local events per peer already result in a large number of global events. All these events have to be stored in the event queue. Especially in structured P2P networks each peer generates a number of periodic events. In order to guarantee a stable overlay and a consistent view of the data, most P2P algorithms involve periodic maintenance overhead. Chord, e.g., uses a periodic stabilize procedure to maintain the ring structure of its overlay as well as a periodic republish mechanism to ensure the redundancy of stored resources. Moreover, since P2P networks are mainly used as information mediators, a simulation usually involves a great number of possibly parallel searches. The choice of an efficient data structure for the event queue is therefore

especially vital to the performance of large scale P2P simulations.

In order to be able to compare two different data structures to each other we need an appropriate measure. The most common measure in this context is the so called "hold time". It is defined as the time it takes to perform a dequeue operation immediately followed by an enqueue operation. Note that the size of the event queue is the same before and after the hold operation. It is easy to see that different data structures have different hold times. A simple sorted list, e.g., has a hold time of $O(n)$, where n is the current size of the event queue. While dequeue operations can be done in $O(1)$ (simply take the first element of the list), an average enqueue operation takes $O(n)$ steps, since the event has to be inserted into the list according to its time stamp. Similar, structures like trees and heaps have an improved hold time of $O(log(n))$.

It has to be noted that, the hold time only states the order of magnitude of a dequeue and an enqueue event. Yet in practice there is a significant difference between say $100 \cdot log(n)$ and $log(n)$.

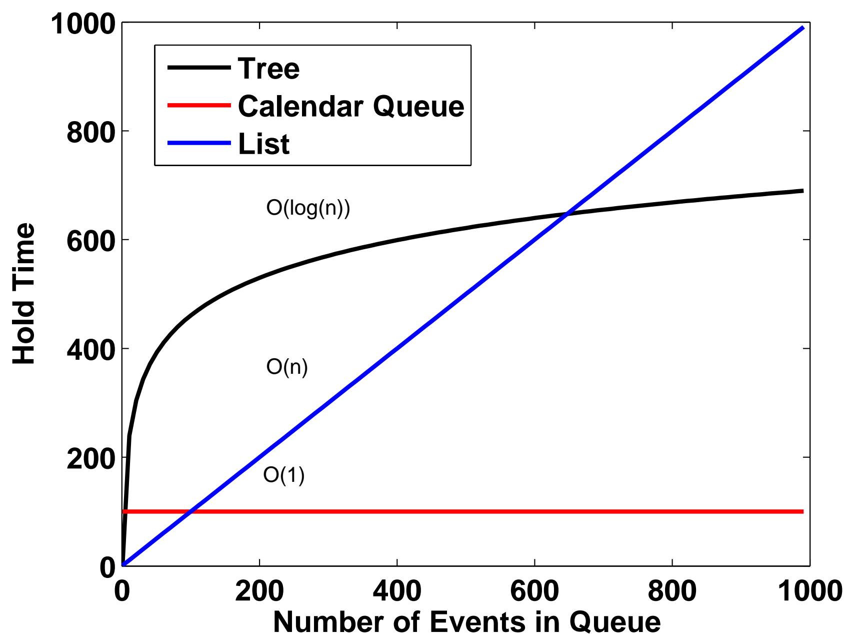


Figure 2: Hold times for different data structures

Especially in small simulations, where there are only a few hundred events, the order of magnitude might not be the only crucial factor. As can be seen in Figure 2, depending on the constant factor, a linear increasing hold time might outperform a logarithmically growing hold time, given that the number of events remains relatively small. In large scale simulations, however, the order of magnitude becomes the determining factor in terms of performance of the event queue. An optimal solution therefore is a data structure with a hold time of $O(1)$ independent of the size of the current event queue. Ideally, this hold time can be achieved without the need for additional computer memory. In the following we therefore summarize the main idea of a calendar queue, a queue with a hold time of $O(1)$ [1]. We discuss its advantages

and disadvantages. In general, the calendar queue can be applied to any large scale simulation. As an example for a priority queue which is designed for a special purpose, we introduce a priority queue with dynamic ranking. This data structure can, e.g., be used to realize the waiting queue of downloading peers in eDonkey. In this context, the incentive mechanism is realized using credit points, that grow with the amount of uploaded bytes. The priority queue for the waiting upload requests is sorted by peer rankings. After successful fragment downloads the credit value is updated. We will draft the corresponding requirements and look into efficient priority queue management.

2.1.1 Calendar Queue

In any discrete event simulation the hold time of the event queue is extremely important as up to 40 percent of the execution time can be spent enqueuing and dequeuing events [2]. There are numerous proposals to realize efficient priority queues [3]. In this section we show how a basic calendar queue [1] with a hold time of $O(1)$ operates. The main advantage besides the hold time is that it is a simple and intuitive data structure. It basically works like a regular desktop calendar. To schedule a future event (enqueue operation), one simply turns to the current day and writes down a corresponding note. In order to find the next pending event (dequeue operation), one starts with the current day and moves from day to day in the calendar until he finds a non-empty calendar day. This procedure describes exactly the way a calendar queue works, except that a year in the calendar queue has a total of N_d days and each of these days consists of T_d time units. The year is realized as an array of size N_d. Technically, a year therefore consists of $T_y = N_d \cdot T_d$ time units. To cope with the situation of more than one event on one day, multiple entries can be stored per day using a simple data structure like a linked list. This list contains all events for that specific day.

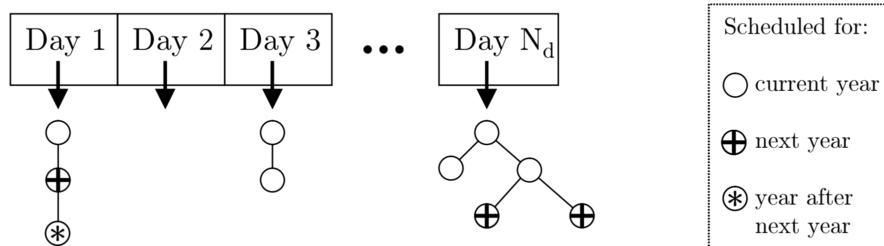


Figure 3: A simple calendar queue

Figure 3 illustrates a simple example of a calendar queue. There are three events on day 1, two events on day 3 and five events on day N_d, the last day of the year. This day also demonstrates that the data structure used for multiple events on one day does not necessarily have to be a linked list. In this example we use a tree like structure for day N_d. Also, note that there does not necessarily have to be an event on each day. There is, e.g., no event scheduled on day 2. To insert a new event into the calendar queue the time stamp of the event is used to calculate the corresponding day on which it should be scheduled. The index of the corresponding day in the

array is computed as

$$index = \left\lfloor \frac{timestamp}{T_d} \right\rfloor + 1 (\mathrm{mod}\ N_d),$$

where $timestamp$ represents the time at which the event is due and the starting index of the array is 1. The event is then added to the corresponding position in the list at this specific day. For events with a time stamp scheduled after day N_d a division modulo N_d is performed to determine the day on the corresponding year. The events marked with a cross could, e.g., be scheduled for next year and the event with the star for the year after next year. To dequeue the next event in line one starts at the array entry corresponding to the current simulation time and moves through the calendar until the first event is found. Thereby, events, which are scheduled for one of the following years, are skipped. Once the final day of the year, day N_d, is reached, the year will be incremented by one and the dequeuing process is resumed at day 1.

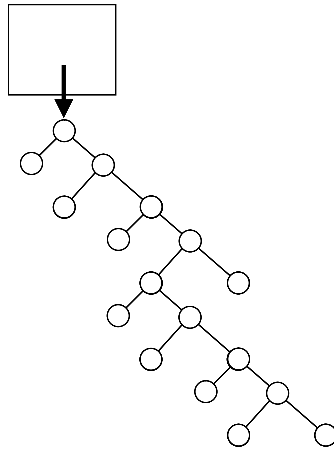


Figure 4: A day with too many events increases the enqueue time.

To achieve a hold time of $O(1)$, the parameters N_d and T_d have to be chosen in such a way, that there are only a few events per day and the majority of events lies within one year. If a day is too long or the number of days is much smaller than the number of events, there will be a large number of events on each day as shown by the overloaded day in Figure 4. Thus, the enqueue time will be excessive because of the time needed to insert an event into the corresponding data structure (cf. the heap in the figure). If, on the other hand, the number of days is much larger than the number of events (cf. Figure 5), the dequeue time will raise, as a lot of days without any event have to be examined until the next event is finally found.

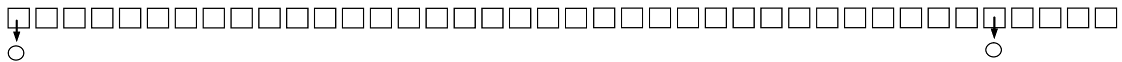


Figure 5: Too many days increases the dequeue time.

In most P2P simulations, the event distribution is not skewed and does not change significantly over time due to periodic events of the participating peers and the like. In this case, it is easy to

predict the number of events per time unit. The length of a day can then be set to a fixed value in such a way that there are few, say about three, events per day and the number of days in such a way that most of the events fall within one year. If, however, the distribution of events is skewed or frequently changes over time, it becomes necessary to dynamically adapt the length of a day and the number of days in a year [4]. An efficient way to restructure the calendar queue on the fly can be found in [2].

2.1.2 Calendar Queue Results

To study P2P specific effects on the calendar queue, we simulate a Kademlia [5] based network consisting of an average of 20000 peers. To generate movement (also known as churn) in the overlay, each participating peer has an exponentially distributed online time with an average of 60 minutes. New peers join according to a Poisson arrival process in order to keep the average number of peers at 20000. The simulator is written in ANSI-C to be as close to the hardware as possible. Based on previous experiences we use a calendar queue with $N_d = 4096$ days where each day is of length $T_d = 100$ms. During the described churn phase a snapshot of the utilization of the calendar queue is taken. Figure 6 shows all 4096 days on the x-axis and the corresponding number of events scheduled at each day on the y-axis.

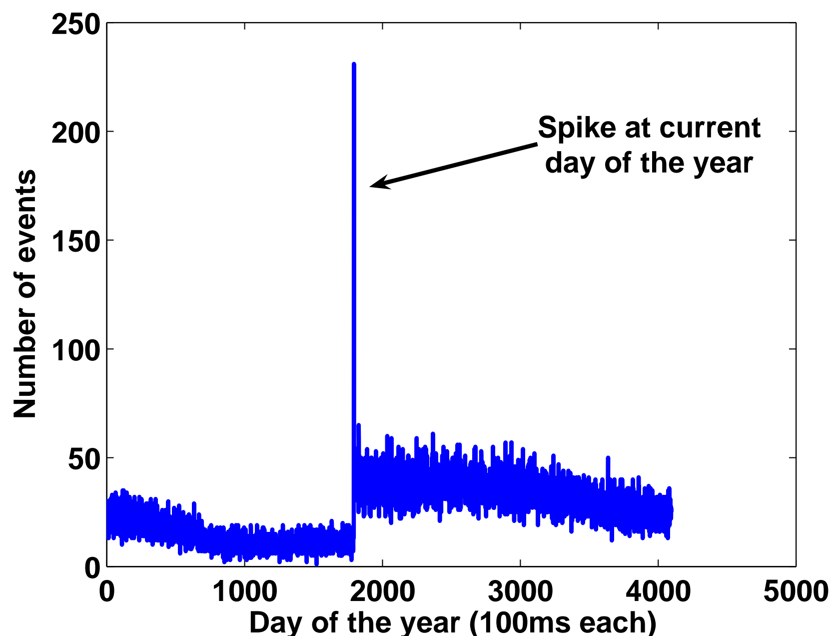


Figure 6: Snapshot of a calendar queue with $N_d = 4096$ and $T_d = 100$ms.

The spike in the figure corresponds to the day on which the snapshot was taken (day 1793 of the current year). All events to the left of this day are scheduled for one of the following years. All events to the right of the current day either take place this year or on one of the following years. There are two important details which can be derived from the figure. On the one hand

there are too many events on a single day in general. On the other hand, there is a huge number of events scheduled for the current day. In general we can distinguish three different kind of events in a P2P simulation:

- Events that take place in the near future, especially those scheduled after one single overlay hop.
- Periodic events, like the stabilize mechanism in Chord.
- Events that take place in a more distant future, like timeouts or bucket refreshes in Kademlia.

In our case the events of the first category are responsible for the spike at the current day, since we use an average network transmission time of 20ms in the corresponding simulation while the length of a day is set to 100ms. The intuitive solution to avoid this spike would be to shorten the length of a day. However, as long as the total number of days remains unaltered, the average number of events per day will remain unaltered as well. Therefore the idea is to shorten the length of a day, while simultaneously increasing the total number of days. From a global point of view there are quiet a number of events at each millisecond in a large P2P network. We therefore decided to first of all shorten the length of a day to just 1ms. The danger in increasing the total number of days N_d is that there might be many days without any event. Since the average number of events per day in Figure 6 is approximately 25 we decided to increase the total number of days to $4096 \cdot 8 = 32768$, resulting in a new average of about 3 events per day. The results of the new run with 32768 days and a length of 1ms per day are illustrated in Figure 7.

As expected, there are approximately 3 events per day now and no burst of events at the current day. Furthermore, periodic and more distant events are equally distributed among all days of the calendar queue. The corresponding values for the parameters T_d and N_d therefore provide a priority queue with a hold time of $O(1)$.

In some situations, however, the adaptation of the parameters is not that easy. For example, an often used method in large scale simulations is to pre-calculate events which correspond to the behavior of the user. That is, events like joins, searches, or leaves, which are triggered by the user and are independent of the applied P2P algorithm, are calculated before the simulation and written to a file. This file is then used as input at the beginning of the simulation. There are some advantages to this approach:

- The event file can be given to different simulators in order to achieve a better comparability of the corresponding results.
- It becomes possible to implement new user models without having to change the simulator in any way.
- Log files and traces of emulations and applications can easily be translated into input files for the simulator.
- The simulation time is slightly reduced due to the pre-calculated events.

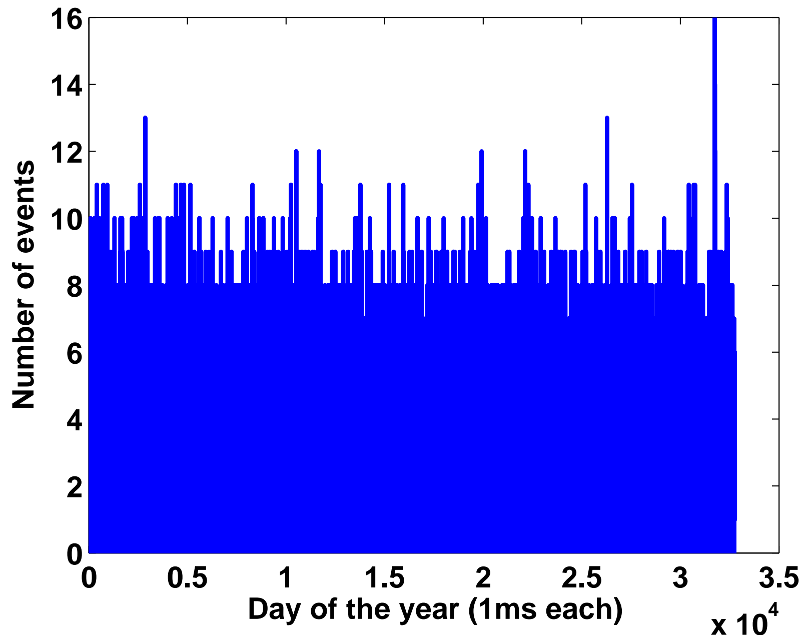


Figure 7: Snapshot of a calendar queue with $N_d = 32768$ and $T_d = 1\text{ms}$.

However there is a big disadvantage in terms of performance of the event queue. Since all user specific events are put into the event queue at the start of the simulation, the number of events per day increases significantly. Figure 8 illustrates this problem in detail. For the sake of clarity we plotted the moving average with a window size of 40.

The blue curve shows the distribution of all events in the event queue. The events can be split into those read from the input file (red curve) and those generated during the simulation (green curve). In this case the increased number of events per day is obviously generated by user specific events. The enqueue time of an event will therefore no longer be in $O(1)$ since there are too many events per day now. A solution to this problem is to maintain two different queues for the two different kind of events. A regular calendar queue for events generated by the simulation and a simple sorted list for the user specific events. With the parameters used in Figure 8 the calendar queue offers a hold time of $O(1)$ for events generated by the simulation. Since user specific events are already sorted in the file, the enqueue operations into the sorted list at the beginning of the simulation can also be done in $O(1)$. There are no more enqueue operations into this queue during the simulation and dequeue operations can be done in $O(1)$ as well. To guarantee the functionality of the double queue concept the dequeue operation is slightly modified. The simulator simply compares the next scheduled events of both queues and executes the one with the smaller time stamp. This way, the advantages mentioned above persist while the management of events remains in $O(1)$.

The above example convincingly motivates the need for special event queues under given circumstances. In the following section we therefore present a specialized priority queue with dynamic ranking, which can, e.g., be used to realize the waiting queue in eDonkey networks.

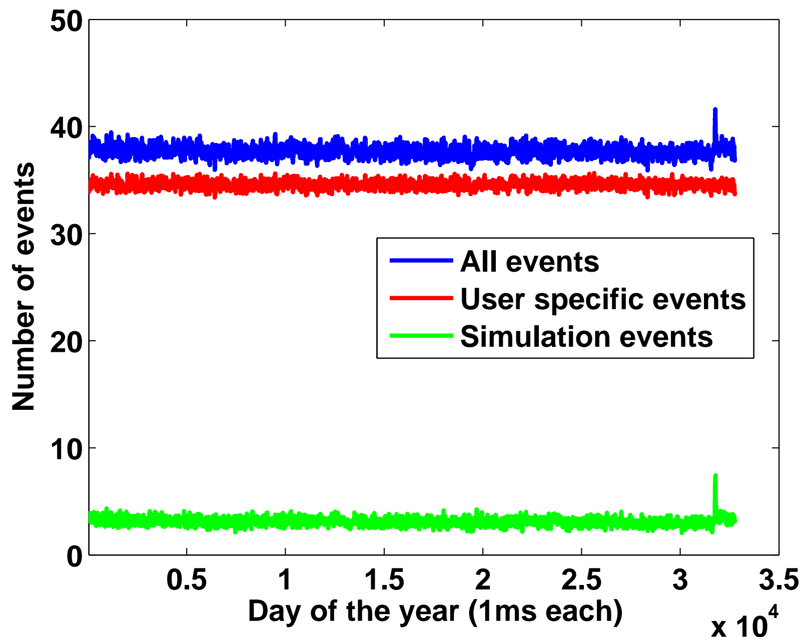


Figure 8: Composition of the events in the calendar queue.

2.1.3 Priority Queues with Dynamic Ranking

P2P systems head towards equal treatment of all peers using incentive mechanisms. When multiple peers request data from a provider, it has to determine bandwidth allocation. Due to the average Internet connection restrictions, only a limited number of peers will be processed at a time. During upload a so-called *upload-slot* is occupied and will be released after completion. Then the next upload to another requestor can be processed.

Because many peers request uploads from a peer, there can be thousands of peers asking to be served first. P2P applications watch up to three properties:

- time: the duration from first contact request to now. Longest duration should be served first.
- resource priority: A peer may share multiple resources, but speed up certain ones, e.g. seldom or newly introduced files. Using file priority, a peer can control bandwidth usage between all shared resources.
- peer priority: Peers can have lasting cooperations and therefore prefer previous partners to unknown peers. This factor realizes incentive mechanisms.

The eDonkey network realizes an incentive mechanism based on these three factors. They are linearily combined and make up the so-called *waiting score*. The next upload slot will be given to the requestor with the maximum waiting score. While file priority and waiting time are mostly static values, the peer priority changes dynamically with any transaction to the corresponding

peer. The *credit points* reflect this change: with every completed upload the credit grows up to a value of 10 (high precedence), while any download consumes some credits, until it reaches the bottom value of 1 (low precendence). This causes that any transaction between two peers influences the sort order of the waiting queue.

At large, the P2P performance an user experiences is made up of picking this peer for upload-slots and it influences directly when a resource download is completed. We consider here an efficient implementation of picking the next peer by maximum waiting score. The requests can be handled in a waiting queue, but resorting this can be expensive. Figure 9 displays for requests and their line up due to their waiting score. It shows that an high priority requests might be served in front of an longer waiting request.

Peer	A	B	C	D
Time	3 min	9 min	3 min	3 min
Peer Priority	10	1	1	1
Res. priority	1	1	1.8	1
Waiting Score	1800	540	324	180

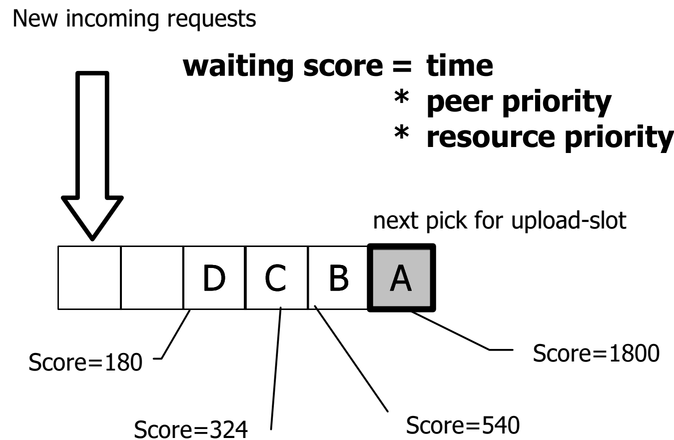


Figure 9: Waiting queue with score computation

The credit point mechanism balances uploads with downloads, such that an active cooperation in both directions is accelerated, but a transfer into one direction only is slowed down. Figure 10 shows the local credit computation with three peers that were download sources for 2, 5 and 10 MBs.

In a large scale simulation, this means intensive work on the data structure. A simple approach would have a queue containing all requests and re-compute the scores for the next pick. Upload-slots in eDonkey are limited to 1 MB or 5 minutes, resulting in $1/5 * m$ picks per minute over all m simulated peers. The queue size n is restricted (e.g. $n = 4096$ in eMule) and in average all waiting slots are filled. By sorting the queue, this approach costs $O(m \cdot n \cdot \log n)$ per simulation minute. This forms a bottleneck w.r.t simulation time and therefore we examine other options.

At first we look closely when and how waiting score changes occur. We assume that the file

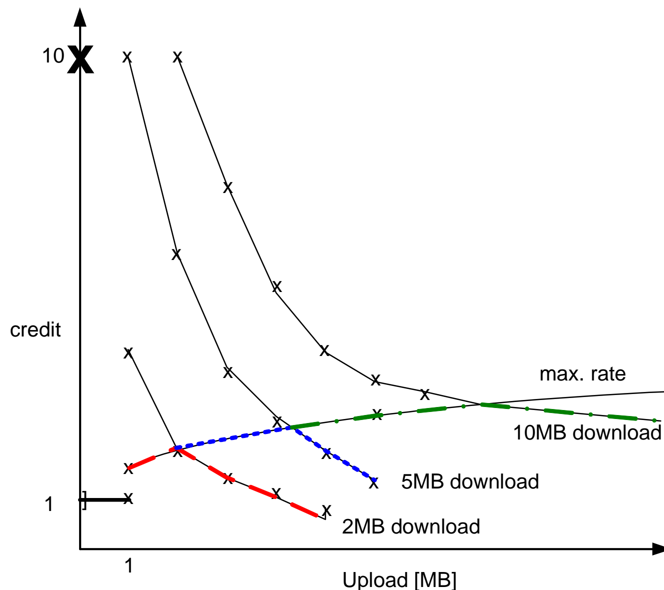


Figure 10: Moving an updated value between sorted runs

priority is constant. The waiting time continuously increases, but it grows simultaneously for all requests. Only the credit is dynamically changing: this is due to parallel operations and its value may increase and decrease, but most updates will be minor changes.

We look into two aspects: First, we investigate the characteristics that accompany re-sorting. When a rescheduling event occurs, either one of the three factors has changed. The massive operation is there is a slight change of the credit (e.g. decrease by 0.1). An example (cf. Figure 11) is the sorting $ABcD$, where ABD are sorted runs with $A < B, B < c, c < D$. If the sort key of entry c is decreased, its new position is somewhat earlier and some other entries will be shifted. The end state is therefore $Ac'BD$ with $A < c', c' < B$. Thus complete reorganization for a single event is not necessary. Finding the correct B and the new position of c' can be found in $O(\log\ n)$ at worst case. An efficient implementation can take advantage from the restricted search field, given from the alteration and the previous position.

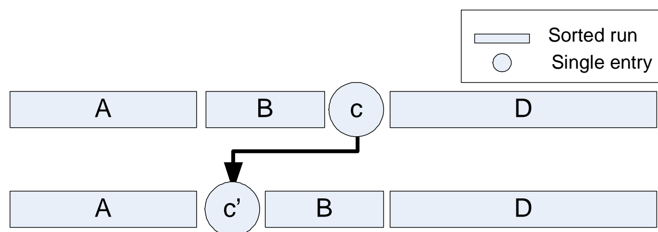


Figure 11: Incentive-driven Credit Computation

This is why we examine second, how the magnitude of entries can be shrunk down. Similar to calendar queuing, we suggest to dissolve this complexity by grouping entries. Note, that there

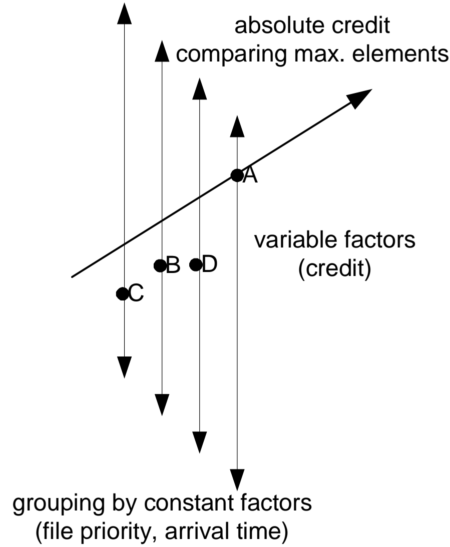


Figure 12: Separating constant and variable ranking influence factors

Job	arrival time	credit value	result order
A	4th	high	1st
B	2nd	-	2nd
C	1rd	low	3rd
D	3rd	high	4th

Table 1: Example download jobs

do not exist disjoint score ranges. Instead, we suggest to distinguish the constant and variable fractions of the score. We can then easily derive the complete queue by projection (which would be expensive). However, finding the top pick is easy, because we can use a another priority queue. We build several queues with a certain range of constant values (cf. Figure 12 with Table 1). When an update occurs, only the variable fraction changes. Because each queue size is much smaller than the previous large queue, queuing speeds up.

2.1.4 Dynamic Ranking Measurements

Our experiemnts investigated an efficient datastructure for score-based waiting queues. Two parameters impact simulation efficiency. The most important influence factor for simulation time is the *maximum number of stored jobs*. This essential size decides about the efficiency of all underlying data structures. Deployed eMule clients limit their size to *4096* and these queues are usually heavy crowded. We assume, that for each processed job the free queue slot will be immediately occupied by a successor.

The second assumption considers ratio of the update frequencies to the turn frequency. For a clear understanding we first sketch both operations. A *turn* picks the top-most element for

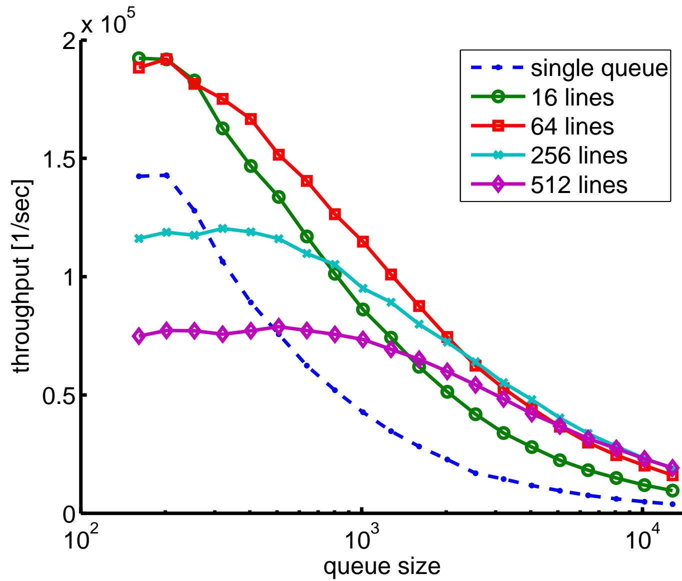


Figure 13: Throughput by varied buffer sizes with quota 1:1

processing and removes it from the queue. The free queue slot is filled with a element that is newly queued according to its peer and resource priority. A completed download triggers an *update* of the peer priority (if there is an pending upload). To reflect the new priority in the queue order, this element must be resorted. Unless otherwise stated we assume a quota of *1:1*, which is the average behavior in small groups. This corresponds to the long-term average in incentive-supporting P2P systems, as any received byte has been sent somewhere else.

Figure 13 compares queue strategies with varied buffer sizes. We measured the throughput of credit point alternations with the corresponding waiting queue update. The traditional single lined queue has exponentially decreasing performance. With large queue sizes this solution is clearly inferior to the other options. Multiple queue lines can accelerate throughput by the factor of two. As a rule of thumb, there should be enough lines to keep the queue size around 25. In the range of 300 up to 2000 queue entries a data structure with 64 lines performs better than with 256, which keep than up for larger queues.

Figure 14 compares varied quotas. When insert operation outweight, there is nearly constant performance. The graph shows, that the lined queues show good behavior with heavy update characteristics. For 4k queue sizes, the 64 line variant gains the best performance.

Summarizing, queue processing with heavy updating can be accelerated. Our solution distinguishes between constant and variable factors and builds lines of similar constant lines. Then updates work on much smaller queues. We showed that lined queuing is superior for all queue sizes and all quotas.

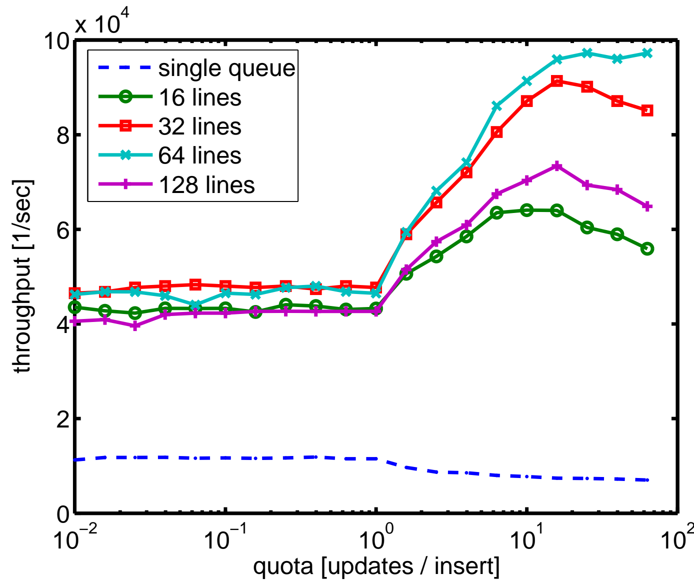


Figure 14: Throughput by quota with queue size 4k

2.2 Event Design Algorithms

The previously discussed performance of the event queue is of course not the only factor, which influences the efficiency of large scale simulations. It is almost equally important that the design of events utilizes the specific features of the queue. The time needed to delete or move events in the queue might, e.g., play a decisive role. In P2P simulations, however, it is often necessary to erase timeout events or to reorganize a large amount of events in the queue. We therefore discuss some possibilities to avoid the corresponding problems and show how to enhance the performance of a large scale simulation using event design algorithms, which are well adapted to queues of discrete event simulation.

2.2.1 Periodic Events

As long as there are only enqueue and dequeue operations on the queue, the performance of the calendar queue is known to be in $O(1)$. However, sometimes there is a need to delete events from the queue, just like a date in real life might get canceled. A possible reason could be a timeout event which is no longer needed or has to be moved to another date. The same is true for already scheduled periodic events of a peer which goes offline. The most obvious way to cope with obsolete events is to search for the event in the queue and delete it. If this has to be done frequently, however, the performance of the event queue degrades significantly. In the worst case the entire calendar has to be searched with a running time of $O(n)$. This process can be sped up by investing some computer memory. For timeouts, e.g., a peer can store a flag indicating whether a search is already done or not. If so, the timeout event can be discarded when being dequeued. Periodic events could also check whether the corresponding peer is still online. Otherwise the periodic event will be discarded as well and started again the next time

the peer goes online. If, however, it is possible for a peer to go offline and online before the next call of the periodic event, the peer ends up having two periodic events instead of just one. Again, investing some computer memory can solve this problem. For each of its periodic events, the peer stores a flag stating whether an instance of this periodic event is scheduled or not. When now a peer goes online again the flag $has_republish = 1$ might, e.g., prevent it from starting a second instance of its periodic republish procedure. This trade-off between computer memory and simulation running time is not always this easy to solve. The following section therefore discusses how to handle dynamic events efficiently.

2.2.2 Dynamic Events

Dynamic events frequently have to be moved in the event queue or might become obsolete in the course of the simulation. To be able to maintain the performance of the event queue it is especially important to find a smart design for those dynamic events. An interesting example

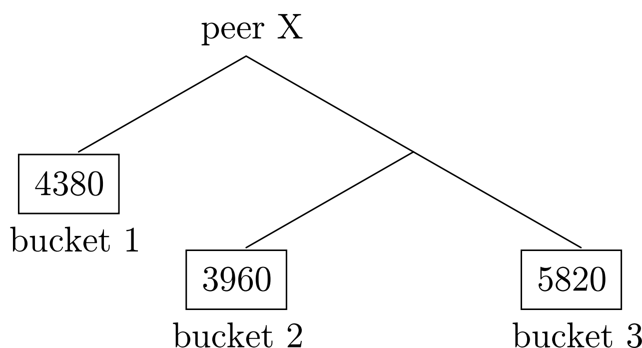


Figure 15: The next refresh times of three exemplary Kademlia buckets.

in this context is the bucket refresh algorithm in Kademlia-based P2P networks. A peer in a Kademlia network maintains approximately $\log_2(n)$ different buckets, where n is the current size of the overlay network. Each of this buckets has to be refreshed as soon as it has not been used for one hour. To guarantee this refresh, a peer maintains a timer for each of its buckets. The timer is reset to one hour every time the peer uses the corresponding bucket, e.g. if it issues a search for a peer or a resource which fits into this bucket.

Figure 15 shows three exemplary buckets for a peer X and the next time they will be refreshed. The next bucket which has to be refreshed is bucket 2 at simulation time 3960. The last bucket to be refreshed is bucket 3 at simulation time 5820. This example can be used to show how to develop a good event design step by step. Assuming we do not want to invest any computer memory, we have to move a bucket refresh event in the queue every time a peer uses the corresponding bucket as illustrated in Figure 16. That is, each time a peer uses one of its buckets for searches and the like, we have to delete the old bucket refresh entry from the queue and add a new entry at the time when the new refresh is due. This, however, increases the execution time drastically, since deleting an event from a calendar queue requires $O(n)$ steps.

To reduce the running time, we should therefore invest some computer memory. For each

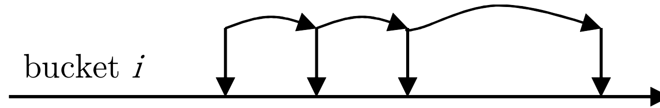


Figure 16: Refresh event moved every time peer uses bucket

bucket of a peer, we could store the time stamp of its next refresh. These time stamps are updated every time a peer uses the corresponding bucket and additionally a new refresh event is inserted into the event queue. Instead of removing the obsolete event from the queue, however, it is simply skipped when being dequeued as indicated by the dotted arrows in Figure 17. That is, every time a refresh event is dequeued, we can compare its time stamp to the time stamp of the next refresh as stored by the peer. A refresh is only executed if the two time stamps match, otherwise the event is obsolete and discarded.

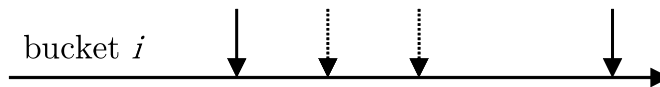


Figure 17: Obsolete refresh events are being skipped

This solution, however, requires more computer memory than actually necessary. Especially if there are a lot of searches in the network and consequently a lot of obsolete refresh events. A more sophisticated solution would be to again memorize the time of the next refresh at the peer, while only using one single event per bucket. Each time the peer uses a bucket, the time stamp of the next refresh is updated locally at the peer. However, there is no new event inserted into the event queue nor is any old entry moved in the event queue. When a refresh event is dequeued its time stamp is compared to the time stamp of the next refresh as stored locally at the peer. If the time stamps match, the refresh is performed otherwise the refresh event is re-inserted at the time of the next bucket refresh as indicated in Figure 18. This way, the memory needed to store the obsolete refresh events can be avoided completely. The problem, however, is that there is still one event for each bucket of each peer. In a Kademlia network of size n, each peer maintains $\log_2(n)$ buckets on average. This still leaves us with a total of $\log_2(n) \cdot n$ refresh events in the event queue. For a peer population of 100000 peers, this adds up to about 1.7 million events!

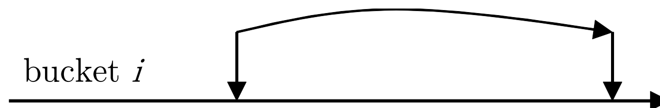


Figure 18: Obsolete refresh events are completely avoided

Considering that bucket refreshes can only be moved forward in time, we can develop an optimized solution in terms of needed memory. As before, we memorize the time of the next refresh for each bucket locally at the peer. This time, however, we only use one single refresh event for the entire peer. This refresh event is scheduled at the minimum of the next refresh times of all buckets of the peer. When dequeued, it calculates the current minimum of all bucket

refresh times and compares it to its own time stamp. Note, that there are only two possibilities now. Either its time stamp is smaller then the current minimum or the two time stamps match. In case of a match the event triggers the refresh of the corresponding bucket. Otherwise, it sets its own time stamp to the current minimum and is re-inserted into the event queue at that specific time as illustrated in Figure 19. Since this procedure takes exactly one hold time, it

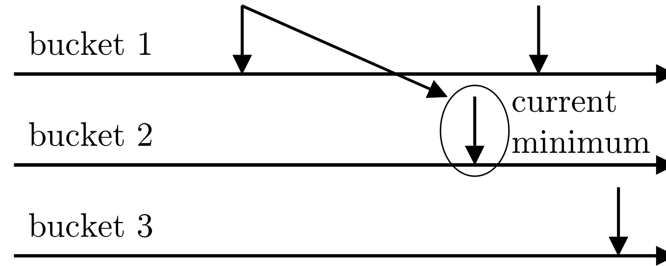


Figure 19: Refresh event scheduled at minimum next refresh times of all buckets

can be done in $O(1)$ for the calendar queue. As an example, consider a refresh event with a time stamp smaller then the current minimum in Figure 15. Comparing its own time stamp, say 3700, to the current minimum 3960 (bucket 2), it recognizes that the refresh it was scheduled for became obsolete. It therefore re-enqueues itself into the calendar queue at time 3960. If none of the buckets is used by the peer, before the refresh event is dequeued again, bucket 2 will be refreshed. The new refresh time of bucket 2 will be set to $3960s + 3600s = 7560s$ and the refresh event scheduled at the current minimum 4380.

2.3 State Representation

To achieve scalability of large scale P2P simulations, the peer cost must also be extremely lean in terms of computational complexity and memory consumption. Therefore, a simplified and compact state representation is essential. In this section we introduce the concept of a process handler, a mechanism, which can be used to reduce the amount of computer memory needed to represent the state of a distributed process. We will also go into the issue of a multi-source download protocol (MSDP). Such a protocol is able to receive content from multiple suppliers simultaneously. For this purpose, a file is divided into fixed length blocks (chunks). Each of these chunks can then be requested from a different uploader. The simulation keeps track of already downloaded chunks of the file. This information is then used to decide which chunk will be requested in the next download process. We establish some assumptions that allow for lean and efficient management of already downloaded chunks at each individual peer.

2.3.1 Process Handlers

As stated above, in large scale P2P simulations computer memory is an almost as equal problem as running time. Due to the highly distributed nature of such systems, however, there are many processes that involve more than one peer. To model those processes each of the participating peers has to store some representation of the process. The resulting copies of the process descrip-

tion at the individual peers are usually highly redundant. We therefore introduce the concept of a process handler to reduce the amount of computer memory needed to represent a distributed process.

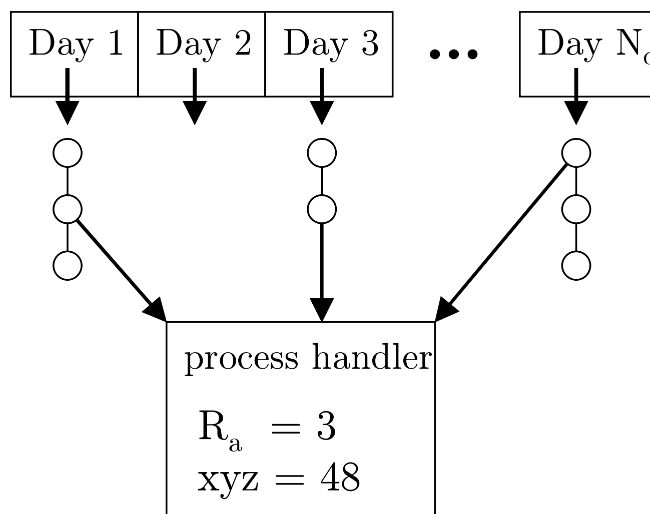


Figure 20: Example of a process handler with 3 remaining accesses

A process handler is a well defined part of the computer memory, where redundant information about a distributed process is stored. Each event or peer participating in the process stores a pointer to the process handler. The process handler includes a variable R_a which determines the number of remaining accesses, i.e. the number of events or peers still pointing to it. Figure 20 shows a process handler with $R_a = 3$ remaining accesses, as there are still three events pointing to it. Each time an event does no longer participate in the process, it decreases the R_a counter by one and deletes its pointer to the process handler. An event, which uses the process handler for the first time accordingly increases the R_a counter by one and stores a pointer to the handler. The last event pointing to the process handler finally frees the memory as soon as it terminates the process. From a global point of view, there are, e.g., many distributed searches in a large scale structured P2P network. Thereby, each search process could be modeled using a search handler. The search handler could store redundant information like the source and the destination of the search, the global timeout of the search, and the number of already received answers.

2.3.2 Modeling Resource Allocation

Content distribution spreads data from some to many locations. Bandwidth and error rates affect the process of completing the transfer and its duration. The intermediate state of nodes is given by the resource allocation. Many protocols divide resources into blocks (i.e. EDonkey chunks), which are validated on a sub-resource level. Locally stored data falls into two categories: *verified blocks* are actually published and redistributed, while *partial data* waits for completion to become verified.

Many protocols use concurrent transfers that head towards completion of disjoint blocks.

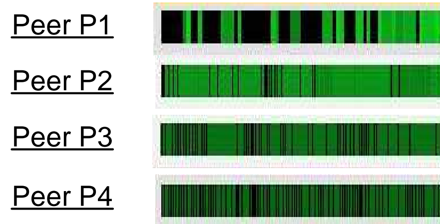


Figure 21: Resource allocation example (darkness = availability, black = local)

Transfers can be aborted at any time and some nodes might store incomplete chunks for a long time. The resulting state is called allocation (cf. Figure 21). Allocation strategies head towards completing partial data and thus share it. However odd allocation could prohibit sharing new fragments and block further downloads [6]. In large-scale simulation, data storage is expensive and must be kept at a minimum. Therefore, we deal with the question how much space will be allocated and how efficient allocation storage is.

Odd fragment distribution can lead to poising downloads [7], caused by a high number of started downloads, but without any complete source. Aberer et. al look into improved data access in P2P networks [8], but do not mention their data structures. Lahmedi et. al examined simulation of dynamic data replication in GRIDs [9], but represent the status as a tree and did not focus on memory effectiveness.

Modeling such effects in large scale simulators is storage-intensive. Example: an EDonkey client that acquires 10 movies results in a total of 1000 blocks. Due to network limitations, received partial data might be heavily fragmented. Modeling the allocation with sixteen-byte-boundaries and a 150 connections with 64 transfer windows causes a worst case of 10 MB status information per peer. This raises the question, what effects can be simulated with alternative models and at what costs. We search for an approximation that maps the current state into an efficient data structure. We will sketch several solutions, determine their storage complexity and discuss weaknesses.

During the upload process two peers negotiate fragments to be transmitted. The transmission might be disturbed either by peer disconnects, packet loss, or session end. The receiving peer acknowledges incoming data for completion check. A session end occurs, when the sender cancels the upload slot to serve another queued receiver. There are several causes to abort a session. In the current eMule implementation a session is restricted by time and data volume.

In the simulation environment, each peer needs to keep track of available fragments. An uploading peer will freely choose a size and offset to be transmitted. Usually a session will not complete a block, so partial data remains until the next upload. Peers will only share blocks that have been verified, i.e. that are completely available at the local site.

- Storing the *percentage* of received data only, is the most simple approach. It is sufficient to model the delay, when no bottlenecks of block sources exist. With the problem of free-riders we cannot assume high block availability. The total cost per peer resource is $O(n)$ for n blocks, with a low constant factor – stored in a word variable. The example would occupy 1 kB per node.

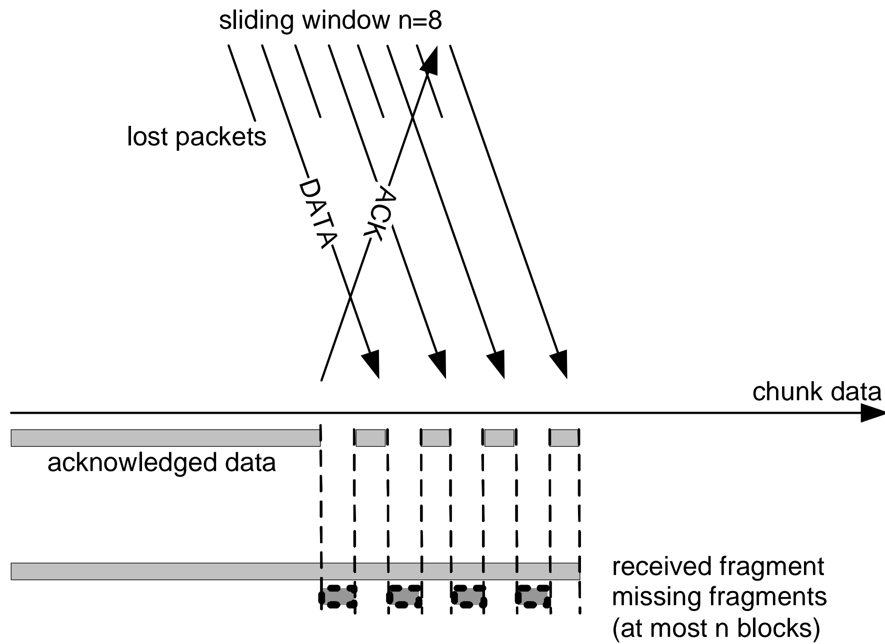


Figure 22: Worst case scenario for chunk availability after session abort

- A more accurate approach uses detailed information restricted on published blocks only. This is the minimum set for download negotiation. It can be achieved either with *range lists* or a bit vector. A range list (of missing blocks) has the advantage that it shrinks towards completion, and as such a huge allocation with few gaps is efficiently represented. Range lists depend on the number of parallel threads, a large number of transmissions can result in discontinuous blocks. The cost is $O(n \cdot m)$ for n blocks and m concurrent transfers. When we assume a limited number of 16 concurrent transfers and a high probability of completing previous blocks, the example request is 32 kB of memory in average. The worst case is that new upload slots cannot complete previously started partial data and the number of partial data blocks increase continuously. In average large numbers of partial filled blocks will occur only at the very end, when all other blocks are completed. While this is a pretty good option, you cannot simulate protocol effects, which we observed in the EDonkey protocol (described later).

- Due to weak implementations some EDonkey implementations (e.g. JMule) create their own fragmentation within blocks. The receiver sends a request containing size and offset of a missing fragment. The sender may choose any size up to the given value. Example: receiver asked for 1500 Byte from offset 0. Sender returns 1450 Bytes only. Correct implementations will fill the 50 Bytes and adjust the maximum packet size. Bad implementations gain partial data with worst case memory allocation. A simulation can investigate this using *range lists on byte level*. The memory usage is expensive: $O(n \cdot k)$ where $k =$ blocksizereceiversize the maximum number of gaps created by a bad implementation (cf. Figure 22). In the example this would be 27 kB per block, with a worst case of 27 MB

status for a single peer. Clearly, this variant is not applicable for large scale simulations.

Study of resource allocation in large-scale systems is crucial to understand the dynamic behavior of these platforms. We have seen that in-depth simulation is intensive, but we can learn from well-behaving peers with much easier methods. Range lists of blocks combined with percentage per block will reveal major effects and allows for collecting data in large-scale simulations. This method accommodates 100,000 peers in 1 GB while allowing four concurrent uploads.

3 Abstractions and Models

The need for different abstraction levels and models for different application scenarios is obvious. On the one hand detailed information is necessary to study the traffic characteristic in the network, thereby measurements below the application layer (e.g. at the transport layer) is inevitable, on the other hand scalability properties are of interest, e.g. during protocol prototyping, and computation complexity should be reduced.

Model	**Abstraction grade**	**Accuracy**	**Computation & memory costs**	**Description**
NS2 simulation	low	high	high	simulate all network layers
Overlay model	middle	middle	middle	model transmission times
Stream approach	high	low	low	streams instead of packets

Table 2: Comparison of different simulation approaches

By adopting different simplifications validity of the simulation has to be ensured. For example, Figure 23 shows the goodput of the download of a file of 540 kB using the Transmission Control Protocol (TCP). While the *throughput* is defined as the number of bits per second transmitted by the source host, the *goodput* represents the number of bits per second received by the destination port. We consider different access bandwidths for different round trip times without packet loss. In that case, the throughput and the goodput are identical. The results are obtained by an ON/OFF simulation with a full implementation of the TCP Reno stack according to 4.4BSD-Lite. The ON/OFF simulation reproduces the packet transfer on TCP/IP layer in order to determine when a user is actively transmitting, i.e. a packet is transmitted over the access network. The packet arrivals for each user are determined exactly according to the TCP/IP stack. The flow control mechanism of TCP tries to adapt the bandwidth to the congestion in the network. This means that TCP's flow control mechanism influences the goodput for downloading a file. Influence factors are the round trip time (RTT), the network access bandwidth of the user, packet losses, and the volume of the file. In Figure 23, the goodput is plotted vs the the access bandwidth of the user in downlink direction, i.e. the maximum throughput physically possible on the particular link, which is denoted as *downlink capacity*. We can see that for some given scenarios (here: for download capacities up to 50 kbps) the goodput approaches the downlink capacity. This means that we do not need to differ between the goodput influenced by TCP and

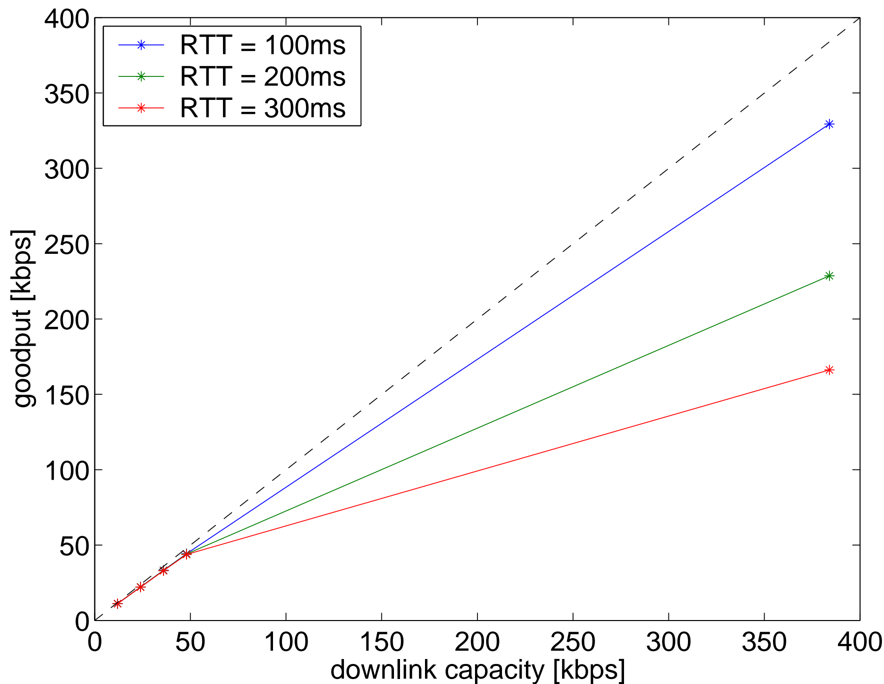


Figure 23: Goodput for the download of a 540 kB file via TCP without packet loss

the downlink capacity. As a consequence, it is not required to simulate TCP if there are no remarkable differences. However, this assumption is not true if the volume of the transmitted file is too low or the downlink capacity is such large that the number of currently transmitted bits on the downlink is below the network's bandwidth-delay product. In that case, a detailed ON/OFF simulation is required for determining the goodput. The resulting mean goodput (of several simulation runs) for such a scenario can then be used for simulations of the same scenario without mimicing TCP's behaviour. This approach reduces the overall computational efforts and enables to generate statistically ensured simulation results.

Figure 23 shows that round trip times have an influence on the throughput. Therefore, it may be necessary to model transmission times for certain applications. Random trip times, e.g. negative-exponentially distributed, are a simple way to model it, but sometimes this is not sufficient. In our current research, we are working on improving searches in P2P networks by e.g. applying proximity neighbor selection. Thus, it is mandatory that transmission times are modeled in more detail. On the other hand, simulating the whole packet-layer would reduce the network size we are able to handle in the simulations.

In the following we present a simulation of BitTorrent on the packet-level using ns-2. Thereby, all details on the different layers are taken into account enabling us to study cross-layer interactions. Section 3.2 presents a coordinates-based transmission time model. Thus, we are still able to apply given transmission times and jitter effects on distinct overlay connections, without the need to model lower network layers. The proposed model may not only be used for P2P simulations, but in all kind of overlay networks. In Section 3.3 we investigate a stream-oriented

approach for a P2P file-sharing network which avoids simulating packet for packet at all. This is possible since in the investigated scenarios the goodput and the downlink capacity are equal (otherwise we could use the mean goodput as downlink capacity). The bandwidth share of eDonkey is used as an example and we approximate the bandwidth that a user gets.

3.1 Packet-based Simulation

BitTorrent is a popular P2P application for a fast dissemination of exactly one file in a P2P network. We investigated the performance of packet-based simulation of the BitTorrent protocol based on the well-known network simulator ns-2 [10].
In BitTorrent each peer controls to whom it uploads data. This is called unchoking. Roughly speaking, the main principle in BitTorrent is that a peer uploads to the four peers with the highest upload rates to it measured every 10 seconds. When a peer has completed the download of the file, unchoking is based on the download rate of the connected peers rather than the upload rate. Details about the BitTorrent protocol can be found in [11].
Since data in BitTorrent is transferred over TCP, one reason to simulate on packet-level is to take the exact behavior of TCP into account. The influence of TCP on BitTorrent's unchoking algorithm can be notable. E.g. the TCP throughput depends on the round trip time (RTT) between the peers. The achieved throughput increases with decreasing RTT. So it can be reasoned that a peer unchokes other peers which are near to it with respect to the delay between the peers. This means that the performance of a peer does not only depend on its upload capacity and the number of peers it knows, but also on the RTT to other peers. To prove or vitiate such cross-layer interactions packet-based simulation is inevitable. Other reasons are to study traffic aggregation caused by P2P applications at the core and edge routers of the network. Furthermore, packet-based simulations can be used to validate abstraction models like that described in Section 3.3.
In the following we present a user and a network topology model to simulate BitTorrent file-sharing networks. These are not only applicable for packet-based simulations. In the last part of this section we present results about the computation complexity of our packet-based simulation with respect to simulation time and memory consumption.

3.1.1 User behavior model

BitTorrent differentiates between two types of peers. On the one hand the leecher, a peer which has completed none or a few pieces of the file. On the other hand the seed, which has completed its download and provides its resources purely altruistically to the network. The lifetime of a peer in BitTorrent is depicted in Figure 24. When a peer enters the network without any completed chunk it has to wait until other peers have uploaded altruistically the data. Then with one or more completed chunks a peer participates actively in the network by also uploading data to others until it finally completes the whole file. For simulation we need to model most importantly the leecher arrival process and the seed leaving process. For a detailed representation also the download pauses of a leecher and its leaving process, which is initiated e.g. by error or by an unsatisfied user and which can depend on the progress of the download, have to be taken into account. Additionally, also seeds rejoin the network although a user has no motivation to do that. This is due to the implementation of specific clients, where the software automatically connects

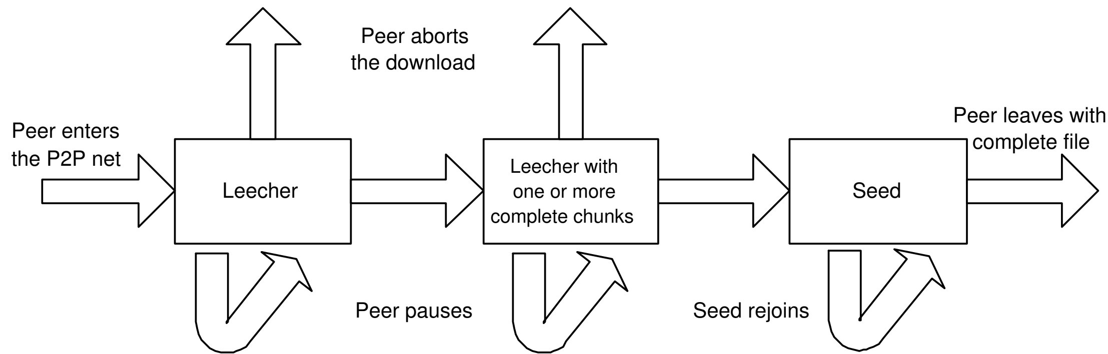


Figure 24: Lifetime of a peer in BitTorrent

to the network after start-up and serves the file. For the sake of simplicity we concentrate in the following on the leecher arrival process and the seed leaving process and neglect the others.
The first approach is to study the worst case scenario for file dissemination, which is called the flash crowd effect. Thereby, initially only one seed and a number of leechers are in the network. This represents an extraordinary burden on the network because only one peer can upload data to the others. In our opinion this is an interesting scenario to study different rules for file dissemination because these protocols must be designed such that the available capacity is used as efficient as possible. This incorporates the case where several peers have nothing or nothing of interest for the others to share.
In the second approach the leecher arrival is modeled according to a Poisson process with mean λ. This can be expanded to a non-stationary Poisson process with different mean values for different intervals in the simulation. By studying real peer populations in BitTorrent networks, e.g. in [12], it was observed that especially during the first days after releasing new content the peer population is much higher than in the following time.
The seed leaving process can also be modeled as worst case and normal case. In the worst case a peer leaves the network immediately after it has completed the download. In the other case a seed stays for a negative exponential distributed time with mean μ before leaving the network. Furthermore, it is highly probable that the original provider of the file also functions as a seed which does not leave the network at all.
A recent study [13] confirmed the assumption of negative exponentially distributed inter-arrival times of peers for their inspected tracker log files of BitTorrent. Furthermore, they observed a power-law distributed lingering time, which is the time a peer stays in the network after it has completed its download. For our investigated small number of peer populations a power-law distribution is not applicable, but should be considered for large and long-lasting simulations.
To model the access speed of an user it should be noted that it does not only depend on the connection type, but also on the willingness-to-share of that user. Most applications offer to limit the capacity of the uplink. Because of asymmetric access lines (e.g. ADSL) and user behavior we assume that the bottleneck in wired networks is the capacity of the uplink of a peer. When altruistic behavior is noticeable also download capacity has to be taken into account since

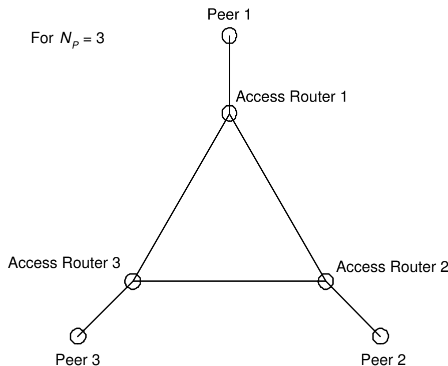


Figure 25: Simplified topology for three peers

in this case a peer receives data from numerous other peers.

3.1.2 Network Topology

Based on the assumption that the bottleneck of the network is at the access lines of the users and not at the routers, we use a simplified topology in our simulations. We model the network with the help of access lines and overlay links. Each peer is connected with an asymmetric line to its access router. All access routers are connected directly to each other modeling only an overlay link. This enables us to simulate different upload and download capacities as well as different end-to-end (e2e) delays between different peers. As an example the simplified topology for a fully-connected peer population of 3 is depicted in Figure 25.
One disadvantage of this model is the required number of links. For N_P peers in the network the number of links in the topology is $Z = (N_P + 1) \cdot N_P/2$. This means, that the number of links increases quadratically with the number of peers in the network causing a memory problem for large peer populations.
One possible solution to overcome this problem is to neglect the differences in e2e delay between the peers. This would result in omitting the overlay links in the already simplified topology resulting in a star topology with $Z' = N_P$ links. A more sophisticated solution for simulations with different e2e delays between peers is presented in Section 3.2.

3.1.3 Simulation complexity

To measure the complexity of the packet-level implementation we ran a simulation of a BitTorrent P2P network for a file of 10 MB. The chunk size was set to the default value of 256 KB. At the beginning of the simulation a flash crowd enters the network, whereas the population was varied from 10-60 peers. Furthermore, only one peer holds the complete file at the start.
We used a Pentium III (Tualatin) PC with 1400 MHz and 2 GB RAM running Debian GNU/Linux 3.0 (Woody) for the simulation. The simulation time for the different number of peers is shown in Figure 26.
An exponential increase of the simulation time can be clearly seen from Figure 26. Whereas the

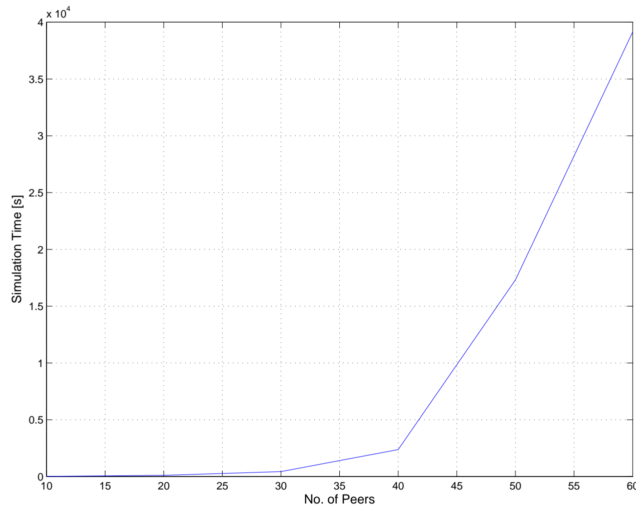


Figure 26: Simulation time for different number of peers

simulations for 40 or less peers can be completed in less than one hour, simulation time for 50 peers is around 5 hours and for 60 peers even around 11 hours. This indicates that packet-level simulation does not scale for even medium sized P2P networks when the file download is simulated.
The memory consumption with this peer populations is uncritically. During simulation of 60 peers only 2.6% of the 2 GB RAM was used.
In spite of the high computation cost, the detailed simulation on packet-level provides an insight into cross-layer interactions and is necessary to validate abstract models for large scale simulations.

3.2 Modeling packet delay with global network coordinates

Our current research on applying P2P mechanisms in Voice-Over-IP (VoIP) solutions leads to two oppositional requirements. On the one hand, we want to simulate a huge worldwide network with hundreds of thousands or even some million participants. On the other hand, the simulations should be as realistic as possible, to be able to develop a protocol that is as very fast and highly reliable. Therefore, it is, amongst other things, necessary to take realistic internet transmission times into account. Participants in different countries and on different continents lead to a wide range of average round-trip times (Table 3) and packet loss values (Table 4)[14]. We think it is, in our area of application, not sufficient to model packet loss and network transmission by using an analytical distribution function (e.g. negative exponential distribution), because lookups in P2P networks can be performed more efficiently if the protocol can relay on proximity neighbor selection [15]. The network transmission model we are going to present in this subsection was mainly developed by Robert Nagel in his diploma thesis. [16].

	Russia	Europe	N America	S Asia	E Asia	Latin America
Russia	55.88	93.46	224.38	502.99	264.64	
Europe	120.58	40.55	155.46	452.87	298.05	256.22
North America	256.75	146.91	50.71	341.67	188.05	258.96
Balkans	130.74	49.17	177.43	480.15	299.09	
South Asia		630.23	414.67	143.85		583.10
East Asia	262.02	326.51	215.80	387.95	34.98	484.65
Latin America	441.90	328.20	297.25	597.74	514.87	81.53
Middle East	234.26	214.83	285.79	504.49	332.19	445.95
Oceania	423.60	398.87	278.88	440.10		
Africa		581.91	724.72			522.37
Baltics	91.01	50.36	173.30	486.21	309.29	
Central Asia		682.71	736.53			
South East Asia		604.19	439.48			
East Europe		148.82	273.96			
Caucasus			692.76			

Table 3: Inter-continental average round-trip times (in milliseconds)

	Russia	Europe	N America	S Asia	E Asia	Latin America
Russia	1.39	0.84	0.52	1.99	0.22	
Europe	1.56	0.27	0.30	0.80	0.30	1.42
North America	2.11	1.11	0.47	0.56	0.34	1.35
Balkans	2.88	3.91	3.07	2.23	3.12	
South Asia		3.68	2.72	3.58		5.95
East Asia	1.29	0.81	0.54	1.16	0.44	9.65
Latin America	1.98	2.56	2.57	2.01	1.01	2.31
Middle East	2.10	1.09	1.70	1.48	1.25	2.40
Oceania	1.67	0.68	0.64	0.55		
Africa		5.05	4.97			3.10
Baltics	1.27	0.08	0.09	0.28	0.12	
Central Asia		4.02	3.10			
South East Asia		5.28	5.96			
East Europe		0.77	1.21			
Caucasus			2.30			

Table 4: Inter-continental packet loss (in percent)

3.2.1 Modeling network transmission times

Table 5 gives a short overview on different approaches to model transmission times for one overlay hop.

Model	Computation cost	Memory	Comment
Analytical function	simple, inexpensive	$O(1)$	no geographical information high jitter unavoidable
Lookup table	simple, inexpensive	$O(N^2)$	high precision problematic data acquisition
Network topology	complex	high	problematic data acquisition
Coordinates-based	inexpensive, expensive offline comp.	$O(N)$	data available good precision

Table 5: Different approaches for modeling network transmission times

The simplest way is to use analytical distribution functions, e.g. negative exponential distributions. They do not require difficult computations and huge amount of memory, but they do not consider the geographical network topology. Worst, a different network transmission time between two nodes is calculated for every packet, and therefore, high jitter values are unavoidable. Thus, packets that are transmitted back-to-back will arrive in a random order.

Storing all inter-node transmission times in a lookup table would lead to very high precision, but this method is not applicable in huge networks, as the size of the table grows quadratically with the number of nodes. Also, acquisition of the data may be very problematic.

Modeling the network topology with routers, autonomous systems and links is a common method to build complex models of the internet, and therefore is applied by many topology generators as Inet-3.0 [17] or BRITE [18]. Yet, drawbacks for using this method could be: it may be problematic to acquire real internet topologies, still a large amount of memory is required for huge networks and the computation of routing paths and transmission times is complex.

We will present a topology model, that is based on network coordinates. It is characterized by a relatively high precision, but low memory and computation costs during the simulation. The required memory scales linear with the number of nodes in the network. The computation of the network coordinates is expensive, but is done offline and the coordinates may be re-used in different simulations. Real internet measurements are available from CAIDA [19] which allows a simulation that is as close to real network conditions as possible. The basic idea is using network coordinates for estimating the transmission time between two nodes. The inter-node transmission time is directly proportional to the geometrical distance in the coordinate space. In Chapter 3.2.2 we describe the "Global Network Positioning GNP" method that we use to construct the coordinate space. Chapter 3.2.3 explains how GNP is used in our simulations and Chapter 3.2.4 shows results that could be obtained by using this network model. We conclude this section with a short outlook.

3.2.2 Global Network Positioning (GNP)

Global Network Positioning [20] was originally developed for predicting packet delays from one host to another. Each node therefore periodically pings a set of *monitors* (or *landmarks*) and measures the required round trip times (RTT). In this section we describe how nodes are able to compute their own position in the geometrical space with this information and the known monitor coordinates.

Creating a new d-dimensional coordinate space at first requires calculating the coordinates of the landmarks. To achieve a high precision, it is suggested to choose landmarks that are as far apart as possible. All round-trip-times between the monitors must be known and the number of monitors n must be greater then the number of dimensions d ($n > d$). The error between the *measured* distance $\hat{t}_{H_1H_2}$ and the *calculated* distance $t_{H_1H_2}$ between the two nodes H_1 and H_2 is defined as:

$$\epsilon(t_{H_1H_2}, \hat{t}_{H_1H_2}) = \left(\frac{t_{H_1H_2} - \hat{t}_{H_1H_2}}{t_{H_1H_2}}\right)^2 \tag{1}$$

The coordinates of the monitors can then be computed by minimizing the following objective function for every monitor M:

$$f_{obj,M}(c_{M_1}, \ldots, c_{M_N}) = \sum_{M_i, M_j \in \{M_1, \ldots, M_N\} | i > j} \epsilon(t_{M_iM_j}, \hat{t}_{M_iM_j}) \tag{2}$$

After measuring the RTT to at least m ($d + 1 < m <= n$) monitors, a node can compute its own coordinates by minimizing the following objective function:

$$f_{obj,H}(c_H) = \sum_{M_i \in \{M_1, \ldots, M_N\}} \epsilon(t_{M_iH}, \hat{t}_{M_iH}) \tag{3}$$

The estimated transmission time $t_{H_1H_2}$ between two arbitrary nodes H_1 and H_2 with coordinates $(c_{H_1,1}, \ldots, c_{H_1,d})$ and $(c_{H_2,1}, \ldots, c_{H_2,d})$ can finally be obtained by computing the geometric distance between the two nodes in the coordinate system:

$$t_{H_1H_2} = \sqrt{(c_{H_1,1} - c_{H_2,1})^2 + \cdots + (c_{H_1,n} - c_{H_2,n})^2} \tag{4}$$

3.2.3 Applying GNP for modeling network transmission

We use GNP coordinates in a slightly different way in combination with ping measurements acquired from CAIDA's skitter project [19]. There are 14 monitors available in the dataset (Table 6), that are mostly positioned at DNS roots. These monitors do daily RTT measurements to a list of selected nodes that are spread over the entire IP space. We are not going to use all monitor nodes for the computation of the coordinates, as good values can already be gained with $d{+}1$ monitors and the computation duration increases significantly if more monitors are used. As mentioned above, it is important to carefully select the monitors. A lot of research has been done in this area [20, 21]. We select our monitors with help of an *maximum separation* algorithm, i.e. we try to select monitors that have a maximized inter-monitor distance (by means of transmission times). This maximization can be solved very easily, as there are only 14 different monitors

available, and it leads to good results. Another promising, but more computation expensive, method is the *Greedy algorithm*, that chooses the set of monitors that minimizes the average distance error (Equation 1) between all monitors.

Monitor name	Location	IP address
arin	Bethesda, MD, US	192.149.252.8
b-root	Marina del Rey, CA, US	129.9.0.109
cam	Cambridge, UK	128.232.97.8
cdg-rssac	Paris, FR	195.83.250.10
d-root	College Park, MD, US	128.8.7.4
e-root	Moffett Field, CA, US	192.203.230.250
i-root	Stockholm, SE	192.36.144.117
ihug	Auckland, NZ	203.109.157.20
k-peer	Amsterdam, NL	193.0.4.51
k-root	London, UK	195.66.241.155
nrt	Tokyo, JP	209.249.139.254
riesling	San Diego, CA, US	192.172.226.24
uoregon	Eugene, OR, US	128.223.162.38
yto	Ottawa, CA	205.189.33.78

Table 6: CAIDA monitor hosts

	b-root	d-root	i-root	k-root	nrt	ihug
b-root		68.882	186.476	172.536	127.812	185.123
d-root	68.882		118.987	95.266	208.739	229.618
i-root	186.476	118.987		36.523	315.139	319.436
k-root	172.536	95.266	36.523		275.874	312.360
nrt	127.812	208.739	315.139	275.874		138.511
ihug	185.123	229.618	319.436	312.360	138.511	

Table 7: Inter-monitor round trip times (in milliseconds)

Table 7 shows the symmetric RTT matrix achieved from a subset of 6 monitors that we use to build a 5-dimensional coordinate space. The monitor's coordinates can now be calculated by minimizing Equation 2 for all monitors.

The skitter data set comprises no inter-node RTT measurements, but it provides us with RTT measurements from each monitor to about 300.000 hosts (Table 8). Coordinates for these hosts can by computed by minimizing Equation 3 for all hosts. This computationally expensive multi-dimensional minimization problem is solved offline. Currently, we are using the Simplex Down-hill Method proposed by Nelder and Mead [22], because it is very easy to implement. Coordinates for the Caida dataset have to be computed once, and can then be reused for all simulations, without any further computation costs. The mean transmission time for the Caida measurements is about 80 milliseconds.

The following paragraph presents the structure of our simulator and the way we integrated GNP. Scenarios we are simulating are described in a *source file*, where parameters like number

	b-root	d-root	i-root	k-root	nrt	ihug
18.166.0.1	84.055	10.535	117.495	85.541	210.628	251.454
81.165.0.1	146.550	85.889	36.159	9.554	284.824	291.408
198.31.255.254	8.777	98.625	177.254	145.013	127.879	196.591
200.63.11.1	249.277	184.413	1060.883	309.182	376.213	523.068
217.200.12.1	172.939	107.576	75.661	27.682	309.860	321.287
...	...	...	...	...	...	...

Table 8: Host-monitor round trip times(in milliseconds)

of total participants, number of online nodes and average online times are set. From it, a traffic generator computes all join, leave and search events, as well as the IDs of nodes and content. We call its output *event file*. The event file can then by put into our *coordinates tool*, that assigns a random host from the Caida dataset to each node in the event file. The tool also adds the appropriate coordinates to the event file. Our simulator automatically detects if coordinates are set or not, and uses the coordinates or a negative-exponential distribution to compute transmission times, respectively. Transmission times between nodes are calculated with Equation 4, but would be constant for each transmission between the same two nodes. Therefore, a log-normal distributed jitter is added to the transmission times, if coordinates are used. This proceeding is based on real internet measurements [23]. A lognormal distribution is denoted as $\lambda(\mu, \sigma^2)$, and its probability density function (PDF) is expressed as:

$$\Phi = \begin{cases} \exp\left(-\frac{1}{2}\left(\frac{\ln(x)-m}{s}\right)^2\right) & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

The parameters m and s can be calculated from measurements where the minimum transmission time θ, the mean transmission time μ and the standard deviation σ are known:

$$m = \ln\left(\frac{(\mu-\theta)^2}{\sqrt{\sigma^2 + (\mu-\theta)^2}}\right) \tag{6}$$

$$s = \sqrt{\ln\left(\left(\frac{\sigma}{\mu-\theta}\right)^2 + 1\right)} \tag{7}$$

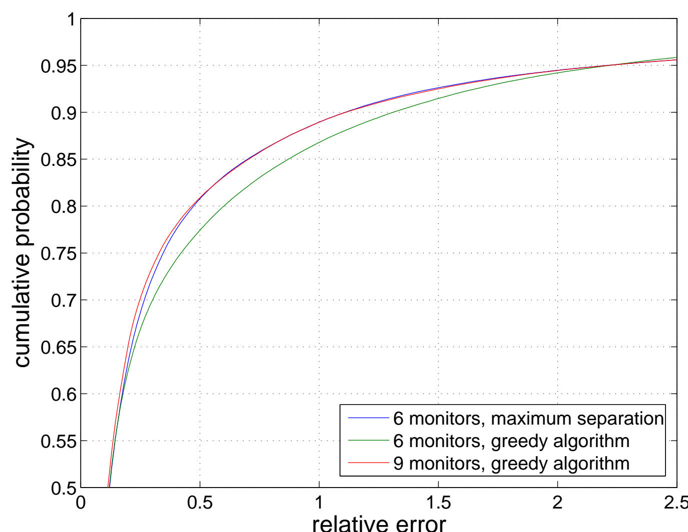
At the moment, we are doing jitter measurements in the Internet to evaluate the parameters $m(\mu)$ and $s(\mu)$ as a function of mean transmission time. We expect narrow lognormal distributions for nodes in close distance and a higher deviation if the foreign node is farther away.

3.2.4 Results

To evaluate the quality of our coordinates, i.e. how exact can we estimate the RTTs between the nodes compared to the real measurements, we use the directional relative error metric:

$$r = \frac{d_{\mathcal{H}_1\mathcal{H}_2} - \hat{d}_{\mathcal{H}_1\mathcal{H}_2}}{\min(d_{\mathcal{H}_1\mathcal{H}_2}, \hat{d}_{\mathcal{H}_1\mathcal{H}_2})} \tag{8}$$

Therefore, we select two monitors, that have not been used to compute the coordinates, and calculate the relative error between them and 2.000 random hosts from our dataset. A directional relative error of plus (minus) one means, that the calculated distance is larger (smaller) by a factor of two than the measured value, whereas a error of zero is a perfect fit. Figure 27 shows the performance of both algorithms. Maximum separation with 6 monitors performs comparably to the Greedy algorithm with 9 monitors. 81% of the calculated round trip times reveal a relative error of less than 50%. On the other hand, 50% of the calculated round trip times have a relative rror of less than 12.3%. We use maximum separation, as it requires significantly less computation effort.



percentile max. sep.	10	20	30	40	50	60	70	80	90
relative error $\leq$ (in %)	1.83	3.84	6.20	8.90	12.35	17.68	26.93	47.57	111.23

Figure 27: Monitor selection method comparison

To evaluate the precision of calculated round trip times with respect to the measured times, we have grouped the measured times and the corresponding calculated times in bins of 50 milliseconds and plotted the directional relative error of each pair on a vertical line (Figure 28). The mean directional relative error is indicated by squares, the 25th and 75th percentile are indicated by the outer whiskers of the line. The figure also shows that GNP performs quite well for distances under 350 milliseconds. A general trend to undershoot in calculated values in apparent; especially for distances of more than 350 milliseconds, GNP undershoots significantly. Still, 93% of all evaluated distances are less than 350 milliseconds, so the influence of significant errors for large distances can be neglected. These large errors result from nodes that are located in areas far apart from the monitor nodes, therefore their coordinates can not be computed precisely.

We are mainly interested in using GNP for calculating transmission times for our simulations. Therefore, we compare the distribution of measured trip times from the Caida dataset to trip times calculated with GNP (Figure 29). The average transmission times is the same

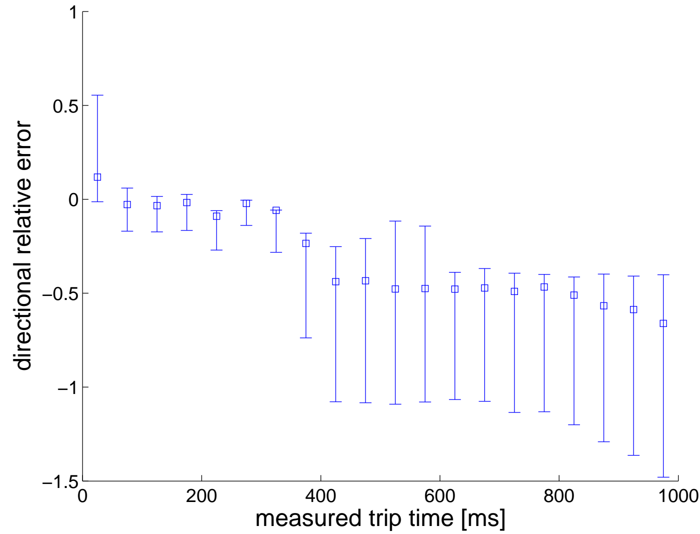


Figure 28: Directional relative error over measured distances

for all curves. The negative-exponential function has a clearly higher standard deviance ($\sigma = 90.99ms$) than the two other distribution based on realisitic topologies, and there are much more very small ($< 25ms$) and large ($> 200ms$) values.

More important, lookups in DHTs are forwarded through the overlay network, until the responsible node for the queried key is found. This results in a series of packets that are sent over the network, with trip times adding up until the lookup is resolved. The sums of these trip times and small additional local processing and queuing delays is the total lookup time. According to the *Central Limit Theorem*, the sum of infinitely many statistically independent random variables has a Gaussian distribution, regardless of the elementary distributions. Figure 30 shows the measured lookup times from simulations with and without using coordinates. As expected, both lookup time distributions are very similar. They look Gaussian, and have approximately the same mean value. The curve corresponding to the negative-exponential distribution is a bit wider, because the standard devince is bit larger for the negative-exponential distribution.

Nevertheless, the network model based on GNP provides us with a more realistic framework, as the transmission time for an overlay hop between two nodes will approximately be the same for all packets, instead of a negative-exponentially distributed random value. Therefore, we are able to apply proximity neighbor selection in our finger and search algorithms. Then, lookup times will be shorter, because a close node can be selected as next hop of a search request. We are expecting a significant left shift of the transmission time curve in Figure 30. Nodes can estimate the transmission times to their neighbors by evaluating existing traffic to this nodes, or by sending active probe packets. Nodes may also predict the distance to another node if network coordinates are applied in the P2P protocol. Network coordinates can be calculated by making use of monitor nodes as it is done with GNP [20] or PCA [24], or by simulating the positions of the nodes with a distributed algorithm like Vivaldi [25, 26]. Meridian [27] is a framework for performing node selection based on network location. We are planning to use

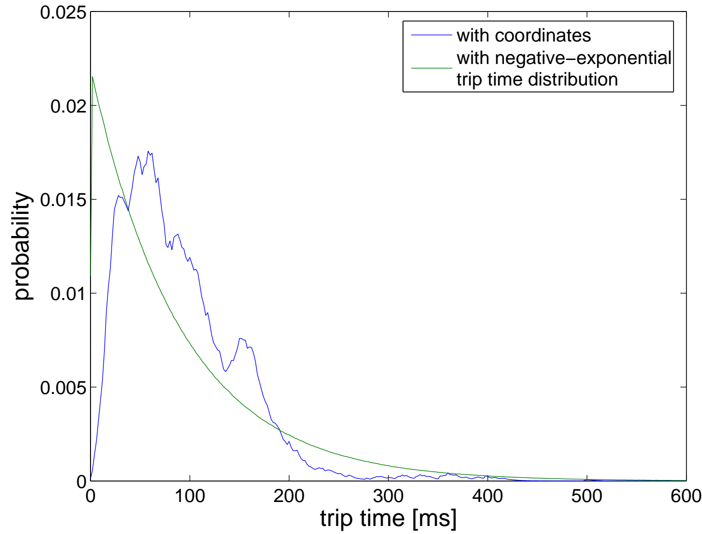


Figure 29: Trip time distributions

Vivaldi coordinates in the P2P protocol, as the algorithm is fully distributed and computationally inexpensive. Therefore, it seems particularly suitable for applying it in P2P networks.

Another interesting phenomena is shwon in Figure 31. If our 5-dimensional coordinates are projected in a 2-dimensional coordinate space, a remarkable amount of clustering can be recognized. If we compare the clusters with a worldmap, even 'continents' may be identified in the coordinates space. This is astoundingly, as coordinates have been calculated from transmission times only. We take this as another fact, that the calculated coordinates are a good representation of the real internet topology.

(a) Worldmap (b) Calculated coordinates

Figure 31: Node distribution in a 2D projection

Concluding we can state, that a topology-based transmission model is a necessary requirement for developing and testing P2P protocols that apply proximity neighbor selection. The coordinates-based method presented in this chapter provides us with a simple, and storage and

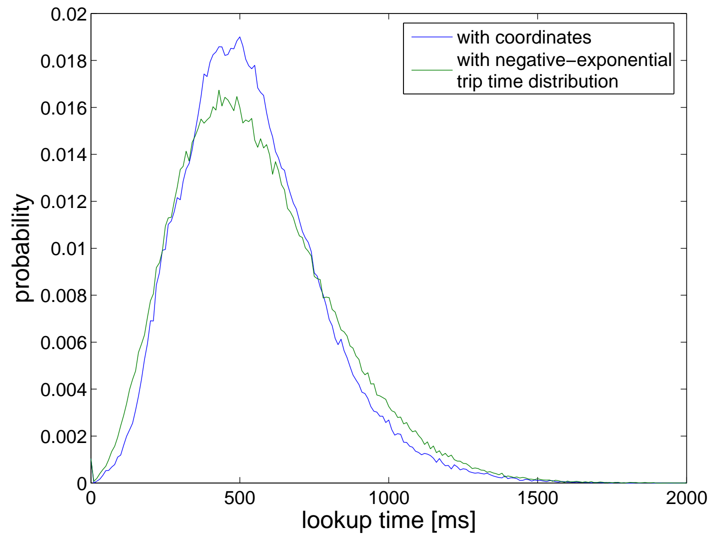


Figure 30: Corresponding lookup time distributions

processing inexpensive way to realistically model network transmisson for overlay hops.

3.3 Periodic and Market-Based Bandwidth Allocation in eDonkey Networks

A main feature of P2P file sharing applications like BitTorrent and eDonkey is the *multiple source download* mode, i.e. peers can issue two or more download requests for the same file to multiple providing clients[1] in parallel and the providing clients can serve the requesting peer simultaneously. The multiple source download is enabled by dividing files into fixed size pieces, named chunks, in order to request parts of the file in parallel. Thus, it is possible for peers to contribute to the P2P network before downloading the complete file.

When an eDonkey client decides to download a file, it asks the providing peers for an upload slot. Upon reception of this download request, the providing client places the request in its waiting list. A download request is served as soon as it obtains an upload slot. A very popular open source implementation of an eDonkey client is the eMule application [28]. The source code shows how the bandwith allocation for the served peers is determined. The eMule client calculates the number of bytes which should be put into the sending buffers of each served peer in the next second according to the bandwidth constraint set by the uploading user. If all peers are able to receive this amount of data everything will stay the same. But if one peer does not acknowledge the data, it will be suspended for the next second. This algorithm guarantees a fair bandwidth split for all peers in the active queue. The active queue is the set of peers that are actually served with data. The maximum size of the active queue is an adjustable parameter of the eMule client.

For every connection the eMule client first creates a buffer for the requested data to avoid delays by loading data from a permanent storage. Now every second the client calculates the

[1] peer and client are used synonymously in this section

available bandwidth in consideration of achieved throughput during the last second and the maximum bandwidth allowed by the user. This value is divided by the number of connections in the active queue. Thereafter this amount of data is enqueued into the sending buffers of the TCP connections for the corresponding peers. Every peer which could not receive the data from the sending buffer within this second will be ignored for the bandwidth allocation in the next second. All peers get the same amount of data as long as everybody is able to receive it. If a group of peers has lower bandwidth capabilities, they will get as much as they are able to receive and the surplus of bandwidth will be shared by the remaining peers due to the same criteria. The resulting bandwidth allocation is called *max-min fair share* [29].

While in Section 3.1 the system is modeled with a packet oriented approach describing the transmission of packets sent from a certain source to its destination. Another option is to consider TCP connections as streams between two peers. In the next sections two models for the stream bandwidth calculations are introduced and compared. The first is the *periodic bandwidth allocation (PBA)*. This model is closely tied to the eMules bandwidth calculation algorithm. Therefore it updates the bandwidth allocation every second. The second approach is called *market based bandwidth allocation (MBBA)*. This model is aligned with the concept of discrete event simulation and avoids periodical updates.

The application of the stream oriented approach neglecting TCP behaviour is possible in our case, since we investigate an eDonkey network in a mobile telecommunication network, like GPRS or UMTS. The pieces of a file which are exchanged between peers using the eMule application have a size of 540 kB. To fully utilize a TCP connection it is necessary that there is a certain amount of data outstanding in the connection. This value can be calculated by the bandwidth delay product and is proportional to the RTT. In [30] Sanchez et al. showed that the RTT is negligible as long as enough data is transferred within the connection, e.g. 400 kB for a GPRS user of coding scheme 4 and multi-slot class 10. Hence the RTT can be neglected.

3.3.1 Periodic Bandwidth Allocation (PBA)

The basic idea of PBA is a straight forward mapping of the eMule algorithm. The algorithm works with local changes of the bandwidth allocation. Before we can focus on the PBA it is necessary to describe the algorithm which computes the fair share bandwidth of a connection at a peer. This **fair share bandwidth assignment** needs the bandwidth which the other peers would regard as fair shared for its calculation. Therefore the **fair share bandwidth for a connection** is defined, which provides this value. These two algorithms work as follows:

Calculation of the fair share bandwidth for a connection

Let Φ be the set of all connections,o the maximum upload capacity set by the user, υ the available, not yet assigned bandwidth, $\eta = \frac{o}{|\Phi|}$ and $\beta : \Phi \mapsto \mathbb{R}$ a map that assigns a connection to its current throughput. The fair share bandwidth ϑ for a connection c is defined by: $\vartheta = \max(\upsilon, \frac{o - \sum_{\mu \in \{\phi \in \Phi : \beta(\phi) < \eta \wedge \phi \neq c\}} \beta(\mu)}{|\{\phi \in \Phi : \beta(\phi) \geq \eta\}|})$
The peer takes a look at the available bandwidth which is yet not assigned to other connections. For all other connections the fair share bandwidth assignment is already done. Then it subtracts the bandwidth of all connections slower than the theoretical mean connection speed, not concerning the connection it was queried for. The rest of the bandwidth is equally distributed to the rest of the connections.

Fair share bandwidth assignment algorithm

With the definitions from above the fair share bandwidth assignment may be defined as a recursive function, as follows in pseudo-code:

function fairshare
(List of not assigned Connections L_{un}, List of assigned connections L_{as}){
    if ($L_{un} = \emptyset$) return
    $x := \infty$
    $\forall c \in L_{un}\{$
        if ($\beta(c) < x$){
            $x := \beta(c), mincon := c$
        }
    }
    if ($x < \frac{leftBW}{|L_{un}|}$){
        assign(c,x), return fairshare($L_{un} \setminus c$,$L_{as} \cup c$)
    }
    else{
        $\forall c \in L_{un}$: assign(c, $\frac{leftBW}{|L_{un}|}$)
    }
}

Example: In order to get a better understanding of the algorithms a short example is provided by Figure 32. We consider there is peer 1 with a network link capacity of 40 kbps and four connected peers. These are named peer 2 to peer 5 and calculate the fair share bandwidth as shown in Figure 32. Thus peer 1 determines the link to peer 2 as the connection with minimal bandwidth, and assigns the 3 kbps for this connection at first. Thereafter it calculates the mean bandwidth for the remaining connections being 12.333 kbps. Peer 3 provided a value of 11 kbps which is lower than the calculated mean and the speed for this connection is set to 11 kbps. Thereafter the new mean value for the remaining peers is 13 kbps. The minimal value of the supposed link speeds is 20 kbps. Thus all remaining connections are calibrated with a speed of 13 kbps, as depicted by Figure 32.

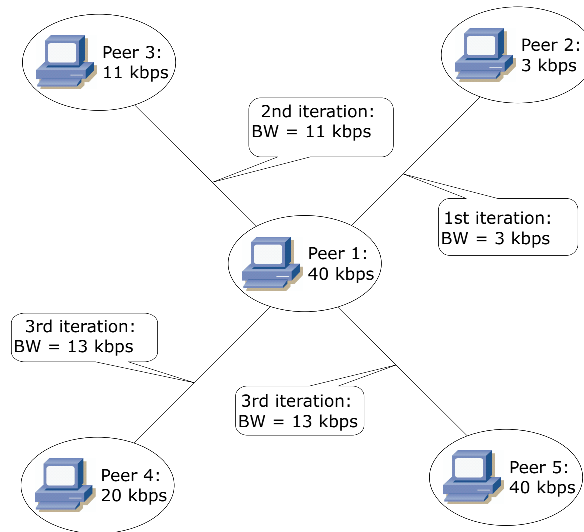


Figure 32: Example for PBA fair share algorithm

The PBA works as follows. Every second all active peers are checked for any changes in their connections to other peers. Either if the number of connections changed or if a connection speed was changed, the PBA fair share algorithm is applied. If a new connection has to be established, this connection is set up with zero bandwidth and the PBA changes the speed to the correct value within the next second. If a connection has to be shut down, it will just be deleted and the PBA will adjust the bandwidth allocation at the affected peers within the next second. This algorithm is easy to implement but has some disadvantages. First, if a connection is shut down the surplus of the bandwidth is distributed to the remaining connections. If any connected peer was already updated in this second this may cause temporal overbooking at the corresponding network link of that peer. Equivalent adding a new connection may cause temporal underbooking of a network link, because the corresponding peer is not able to request more bandwidth from it's other sources. However the influence on the numerical results is small because these two inaccuracies cancel each other and the time period of overbooking / underbooking is rather short compared to the overall length of a connection. E.g. the transfer of standard download unit, which is 540KB in size, takes about $\frac{540*1024*8\,bit}{3000\,bit\,s^{-1}} \approx 24.5\,min$.

´ It has to be noted again that the PBA algorithm models the bandwidth between all peers in the network in order to achieve max-min fair share. If we talk about bandwidth allocation this means that in our model the bandwidth is assigned in such a way that the bandwidth of a peer's upload/download connection follows the max-min fair share principle. As a result of the PBA algorithm max-min fairness is modeled.

Figure 33 visualizes how bandwidth allocation may spread through the network in the worst case. For example peer 1 has shut down one downlink connection. Therefore it tries to re-assemble the leak of bandwidth and asks the connected peers to speed up their connections. We consider all connected peers are able to speed up these connections and all connections marked

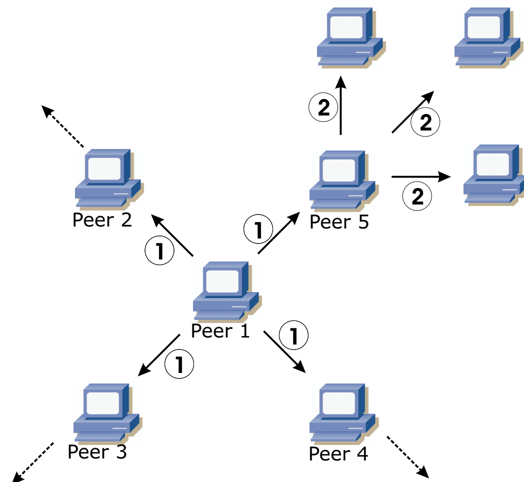


Figure 33: Bandwidth reallocation spreading through the network

with "1" in Figure 33 change their speed. Thus one second later all connections of the connected peers (marked as "2" in Figure 33) may possibly be changed. The connections denoted with a "1" are also recalculated in the second step, but they do not change, because each of them has already applied the fair share bandwidth.Therefore the bandwidth algorithm does not update the same connections more than one time in two subsequent seconds. But cyclic net structures may cause the bandwidth allocation to oscillate. Figure 34 visualizes such a cyclic network structure. Each connection is augmented with a sign representing the trend for the connection change, and the delay of the fair share reassignment.

In Figure 34 peer 1 shuts down a connection at the time t_0. At the time $t_0 + 1$, the connections attached to peer 1 is re-calibrated. We consider the connection between peer 5 and peer 1 is not able to gain a higher bandwidth because of the other links of peer 5. Thus only the connection between peer 1 and peer 2 is changed and the complete surplus of bandwidth may be added to this connection. Now the re-calibration spreads through the network. At time $t_0 + 2$ the connection between peer 2 and peer 3 is slowed down. Peer 3 and peer 4 speed up their connection thereafter at time $t_0 + 3$. This may cause the connection between peer 4 and peer 5 to slow down at time t_0+4. Now the problem becomes obvious. Peer 5 reallocates its bandwidth at the time t_0+5 and the connection between peer 4 and peer 5 was decelerated before. Now peer 5 is able to speed up the connection between himself and peer 1, but peer 1 has assigned the complete bandwidth it freed by closing the other connection to the link between peer 2 and itself. This allocation was fair share at the time $t_0 + 1$ but is not at the time $t_0 + 5$. Therefore peer 1 has to recalculate its bandwidth sharing at time $t_0 + 6$, which will again affect the other peers in the circle.

Although the bandwidth at real eDonkey clients oscillates, our model aims at a perfect max-min fair share bandwidth allocation in an efficient way. Thus, the oscillation may be regarded as a disadvantage with respect to computation time. The periodic character has some drawbacks concerning discrete time simulation. A discrete time simulation normally steps from one event

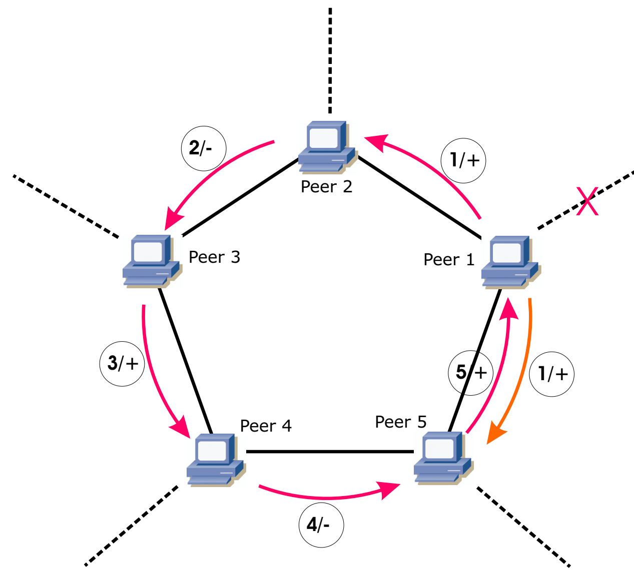


Figure 34: Cyclic dependencies may cause network bandwidth oscillate

to the next. Thus it saves computation time by disregarding time intervals in which no changes occur. It has to be noted, that the PBA introduces additional events and therefore slows down the simulation runtime, if the system state does not change over a longer period of time, e.g. for 1 minute. The second approach which will be described in the next section tries to avoid these additional events.

3.3.2 Marked Based Bandwidth Allocation (MBBA)

In the last section the PBA algorithm for modeling max-min fair share in a network was explained. The algorithm was executed periodically for each peer such that for all peers in the network the connection's bandwidth in uplink and downlink connection follows the max-min fair share principle. In order to avoid periodic bandwidth recalculations it is necessary to reassign the bandwidth of all connections that are influenced. Thus the first step of the MBBA is to collect all connections that may need to be updated. It is not necessary to make a list of the connected peers. Instead it is sufficient to make a list of the network links that are involved. Thereafter the MBBA recomputes all connection speeds in such a way, that fair share can be guaranteed. This is done by an auction like bidding system. The peers' bids are calculated equivalent due to the fair share bandwidth assignment at the PBA. And like above only if one peer's bid is the smallest bid for the connection and the connected peer keeps this bid, they are in agreement.

Marked based bandwidth allocation algorithm

The algorithm iterates through all network links and assigns new bids for the connections (GenerateNewBids). If a network link has assigned new bandwidth to all its connections it will return true and will be stored in a list for removal. Thereafter every connection is checked if it satisfies the constraints for a deal. If the links are in agreement, the connection will set its speed and return true, to be collected in another list for removal. Finally all network links and connections of the removal lists are discarded and set flags on these entities will be deleted.
Flood the connected component and collect all network links in L_p and connections in L_c, label them with a re-calibrating flag and initialize bids to $\frac{link\,capacity}{number\,of\,connetions}$
while($L_c \neq \emptyset$){
    $D_p := \emptyset$ // set of peer that can be deleted
    $D_c := \emptyset$ // set of connections that can be deleted
    $\forall p \in L_p$: if p.GenerateNewBids() then $D_p := D_p \cup p$
    $\forall c \in L_c$: if c.CheckStatus() then $D_c := D_c \cup c$
    $\forall p \in D_p$: remove flag from p
    $\forall c \in D_c$: remove flag from c
    $L_p := L_p \setminus D_p$
    $L_c := L_c \setminus D_c$
}
$\forall p \in L_p$: remove label from p

The algorithm for the bidding is a little tricky. Basically the connections with the lowest bids are filtered. Next the theoretical mean connection speed is calculated. If the minimum speed is below or equal to the theoretical mean connection speed, those bids are kept, and the remaining bandwidth is equally shared by the other connections. If the minimum speed exceeds the theoretical mean connection speed, all connections are assigned with a bid according to the theoretical mean connection speed. Two modifications are applied in order to prevent oscillation and low bids for the last connection of one link. First, the bid for the minimal link is not lowered. This prevents that connected network links that both assume their connection as minimal exchange their speed bids and an infinite loop happens. E.g. network link 1 and network link 2 are connected. Both found the connection to be the minimal link, but network link 2 has bid value x and network link 1 has set his bid to y. x and y are not equal and are both below the theoretical mean connection speed. If the first rule did not prohibit lowering the minimal bid the bidding in the next turn would be exactly the same as before, only that the bid of network link 1 would bid x and network link 2 would set its bid to y. Thus an infinite loop results. The second modification is to prevent lower capacities on links last assigned. In order to guarantee the usage of the complete bandwidth of a network link, it is neccessary to assign the remaining average connection speed to all the connections, if all actual bids are below this value. E.g. there is a high bandwidth network link (NL1) connected to at least three low capacity links and one high capacity link (NL2) with no other connections. The three low ones differ in their bids. Then NL1 selects the minimum bid, adopts its bid according to this link and distributes the remaining

bandwidth equally to the remaining links. Without the second modification NL2 would adopt its bid to the lower bid of NL1. If the bandwidth assignment of NL1 determines the speed of the slow connections in the same round, whereupon minimal bids of the other network links are accepted, the bid for the connection to NL2 does not contain the bandwidth that is not assigned at this moment but is bid for the other connections. However NL2 keeps this bid. In the next round NL1 and NL2 would assign this speed to the connection, because none of them knows about the unused capacity of the other link. This would lead to unused bandwidth in the network. This can also be observed in detail in the example provided on page 45. Peer 8 in the example corresponds to NL2 and peer 2 corresponds to NL1. If peer 8 in the example would adopt its bid to the bids of peer 2, then the link speed would be set to 31.666 instead of 36.666.

Peer::GenerateNewBids

Let Φ be the set of all connections, Ψ the set of connections marked with a re-calibration label, o the maximal available bandwidth, and $\beta : \Phi \mapsto \mathbb{R}$ a map that assigns a connection to its actual speed bid of the other peer. With these definitions the GenerateNewBids procedure can be described as folows:

if ($|\Psi| = 0$) return true
$C := \{c \in \Psi | \forall j \in \Psi : \beta(j) \geq \beta(c)\}$
$x := \frac{o - \Sigma_{x \in (\Phi \setminus \Psi)} \beta(x)}{|\Psi|}$ //the mean remaining bandwidth
if ($|C| = |\Psi|$ or $|\{c \in \Psi : \beta(c) \leq x\} = |\Psi|$)
    min := x
    otherspeed :=x
else{
    if ($(\beta(c)) < x, c \in C$){
        min := $\beta(c), c \in C$
        otherspeed := $\frac{o - \Sigma_{x \in (\Phi \setminus \Psi)} \beta(x)}{\Phi - (|\Phi \setminus \Psi| + |C|)}$
    }
    else{
        min := x
        otherspeed :=x
    }
$\forall c \in C$: bid(c):= min, set minimal flag for c
$\forall c \in (\Phi \setminus \Psi)$: bid(c):= otherspeed
}

Against the other parts of the algorithm CheckStatus is rather simple. The connection speed will be set if the lower bid is marked as minimal and is repeated.

Connection::CheckStatus

```
if (the lower bid is marked as minimal and is repeated){
        set connection speed to the lower bid
        return true
}
else{
        return false
}
```

Example: In order to explain the algorithm in detail, Figure 35 shows a small network example on which the algorithm is applied. Figure 36 depicts the calculated fair share bandwidth allocation.

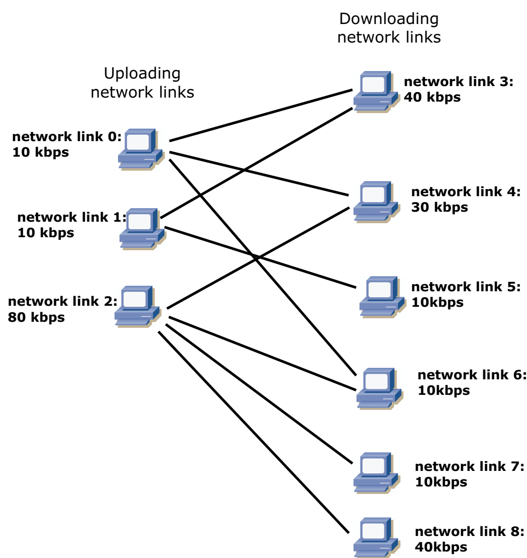


Figure 35: Example for the market based bandwidth allocation

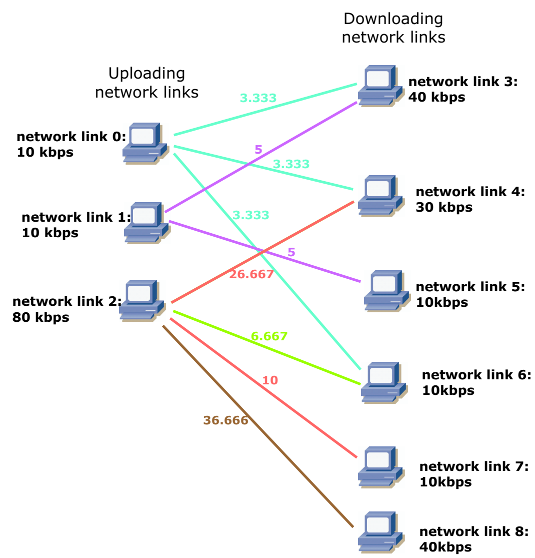


Figure 36: Fair share bandwidth allocation for the example network

During the initial flooding every network link bids its capacity divided by the number of active connections. Thus network link 0 sets its bids to 3.333 kbps, network link 1 and network link 6 bid 5 kbps and so on. In the first round network link 1 and network link 2 notice that all bids of the other peers are higher than their possibilities for fair sharing and therefore keep up there bids. Network link 8 keeps up its bid, because the last minimum connection rule applies for the connection. All other peers find a minimum bid, which is below their fair share possibilities and assign this value to the corresponding connection. Thus network link 0 and network link 1

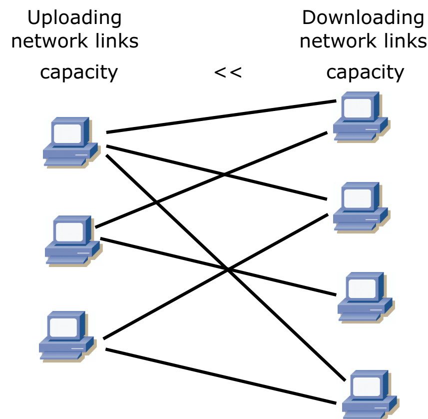


Figure 37: Bipartite character of the network graph

can allocate fair share bandwidth for all its connections within the first turn. In the following the algorithm is applied to the other network links until a solution is found. It has to be noticed that peer 8 shows the need of the last minimum connection rule as mentioned above. It has to be noted, that the uplink and the downlink of a peer do not influence each other. Therefore the network graph is always bipartite, because uplinks are only connected to downlinks and the other way round. A peer can belong to both of the partitions, if it uploads and downloads data at the same time.

However the uplink of a peer is never connected to its downlink, because the peer does only request data it does not have. Thus the network will always look like Figure 37. Due to this restriction and to the fact, that the upload capacity is rather small compared to the download capacity, further optimization of the algorithm can be applied. A connection with a capacity value below the actual and foreseen mean link speed is restricted by the other peer. Therefore it is not necessary to re-calculate its capacity and flooding through this link is not required. Also nodes, which use only such a small part of their capacity that a new connection can not exceed this volume, do not need to propagate flooding. The re-calculation will not change anything in the other network parts. Finally if a link is shut down the re-calibration is called for each of the further connected peers. This could be avoided if the algorithm would recognize that both peers belong to one connected component.

3.3.3 Comparison between PBA and MBBA

In the previous sections two models for the bandwidth allocation were described in detail, the periodic bandwidth allocation (BPA) algorithm and the market-based bandwidth allocation (MBBA) algorithm. In order to compare these two algorithms we consider five long simulation runs containing each about 35.000 completed downloads. The download time depends heavily on all other downloads before, because on the one hand downloads of the same file increase the number of sources but on the other hand downloads of other files may reduce the number of sources according to the file replacement strategy. Additionally the random streams for

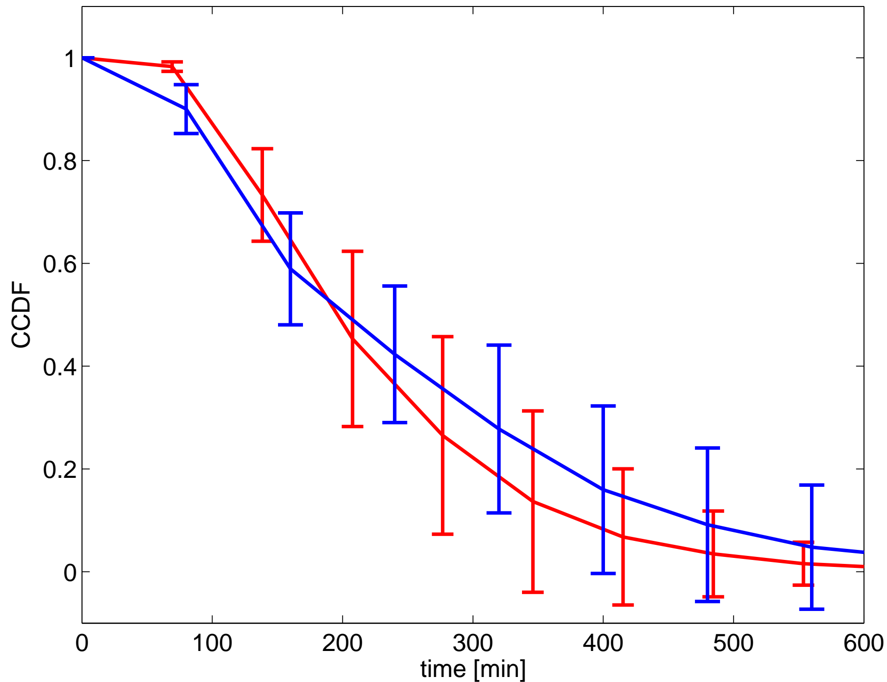


Figure 38: CCDFs of the download time for a mp3 file of 5 MB with a significance level of 95%

generating events in time differ and therefore it is impossible to avoid small differences in the complementary cumulative distribution functions (CCDFs) of the download time for both algorithms. Figure 38 depicts the CCDFs with confidence intervals at a significance niveau of 95%. Both curves stay within the confidence intervals of the other curve. We can conclude that both algorithms are equivalent and produce similar simulation results. For too small files, the stream-oriented approach with neglecting the TCP behavior for modeling the bandwidth of peers in the eDonkey network is not applicable, as depicted in Figure 23. Furthermore, the download times of smaller files are heavily dependent on the waiting time, i.e. if there are serving peers with short waiting queues. E.g. if in one simulation there are some peers with short waiting queues and in the other there are none, the experienced download time differs much. Additionally the bandwidth oscillations of the PBA effects the download of the small files much more than the downloads of files with a larger size. Although this can be avoided by decreasing the time interval between two executions of the PBA algorithm, this would result in a significantly high computational time. Nevertheless, much more simulation runs are needed to get acceptable small confidence intervals for the smaller file classes, which requires again enormous computational costs.

Anyway the computational costs of both algorithms differ a lot. In situation in which several file requests are sent within one second and the number peers in each connected component is large, the PBA is faster than the MBBA. The PBA updates the connection speed of the network links each second and saves therefore computations. Additionally the high connectivity causes the MBBA to recalculate some connections more often than needed. On the other hand, if there

are only a few requests each minute and the ratio of uplink bandwidth to downlink bandwidth is small, the MBBA has a better computational performance.

A simulation in which the stream-oriented approach can be applied can be further optimized with respect to computational time by using a dynamic model. During a simulation run, the bandwidth allocation algorithm which requires less computational time should be used. The decision whether the PBA or the MBBA algorithm is more efficient simply depends on the number of events per time unit; we only take into account events which change the current bandwidth allocation of at least a single network link. However, the problem is that this number of events for the next time unit has to be estimated a priori in order to increase simulation speed. Certainly, the method for estimating this number must not be very complex, as we would increase again computation time and loose the benefit of switching between the PBA and MBBA algorithm on the fly. This task is very interesting and will be investigated in future.

4 Efficient Programming and Parallel Simulation

Ivan Dedinski: One of the main challenges of parallel discrete event simulators (PDES) is the efficient synchronization of logical processes. Synchronization is needed, since if the local time of two processes drifts away, the process with the higher time could send a “message from the future” to the other process. This kind of errors cannot be tolerated by any discrete event simulation, since they can cause effects, that can not happen in a real environment. Efficient synchronization in PDES is not a trivial task. The potential for optimizations without knowledge of the nature of the application are very limited. On the other hand, for some kinds of applications it is possible to gain a significant speedup by using application specific optimizations. This chapter presents an optimization approach suitable for simulating a variety of network topologies, typically spanned by P2P applications.

4.1 Motivation and Related Work

The simulation of P2P overlays has some specific requirements and properties that need to be considered. The most obvious one is that a P2P simulation is only meaningful for a large number of peers. The higher the number of simulated peers, the higher the significance and reliability of the results. Another important feature, that is making parallel simulations for P2P attractive, is the autonomy of the P2P agents. That means, that they will act fairly independently from each other and thus provide a parallelization potential.

Todays research on PDES often concentrates on optimizations in terms of execution speed. However a PDES would also make resources other than CPU power available to the simulated P2P network. A common problem that arises for bigger simulation scenarios is the memory usage. Todays Java machines have the limitation of 2G memory. But even if this constraint would be removed, one still would need a powerful and expensive server machine to provide the required memory banks for large and complex experiments. The PDES makes the usage of already available resources possible like CIP pools, or already existing department labs, that are normally unused out of work times. P2P simulation scenarios could greatly benefit, if using these resources.

Much research on PDES has been done, mainly on the speed optimization by increasing the amount of parallelization in the system. The main problem here is the so called Local Causality Constraint: A discrete event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging time stamped messages, obeys the local causality constraint if and only if each LP processes events in nondecreasing time stamp order. Obeying to this constraint will definitely prevent causality errors, but would not always be necessary, thus causing certain events, which can be executed in parallel, to be executed sequentially.

There are two main classes of PDES mechanisms, that try to increase the parallelism in a simulation. Conservative approaches definitely avoid causality errors, but try to detect events, which can be executed in parallel, in advance, so that no bottlenecks appear in the system. Optimistic approaches on the other hand discover causality errors that have already happened and recover from them. [31] provides a comprehensive study on the different mechanisms of these two classes. One property of all described mechanisms is their general purpose. They are mechanisms applicable to all kinds of simulation scenarios. This chapter introduces a new conservative PDES method that uses knowledge about the simulation scenario and also may influence the simulation up to a certain predefined degree (which implies the non-repeatability of the scenario). On the other hand it may greatly outperform generic conservative and optimistic mechanisms and is simple and verifiable.

4.2 Simulation architecture (requirements)

Developing a P2P Simulation that should consider the specific properties of the simulated network, one needs to design its topology first. Afterwards a partitioning of the topology and a mapping to simulator machines (or LPs) has to be made. The choice of the topology, the partitioning and the LP mapping influence the performance of a parallel simulation to a high degree. One should mainly consider two points: the LPs should be equally (as equal as possible) loaded and the inter-LP communication should be kept at minimum. In this chapter we only discuss static topology partitioning. Dynamic partitioning has much higher complexity and is not covered.

For our P2P simulations we used a topology consisting of some small scale, highly interconnected networks of equal or at least similar size. The links in these networks may have high capacities and small delays. The networks are interconnected with links, that should have low capacities, and delays as high as possible. The partitioning and mapping of the topology to simulator LPs is straightforward and shown on Figure 39 Every small-scale interconnected network is mapped on a single simulator LP, the logical links between the small-scale networks are mapped to physical links between the corresponding LPs. Higher delays on the LP-interconnecting links improve the simulation performance, since the LPs need to synchronize less frequently. Note, that even if a LP-link is not used at all, synchronization is necessary, because of the possibility of sending a packet, which could lead to a causality error. The frequent usage of a LP-link would cause real network traffic, it will however reduce the need of transferring null messages.

This topology construction approach is well suited to model todays Internet, where total interconnection can be found on the backbone, but the edge networks are mainly organized in a hierarchic manner. It is possible to construct, exactly as in the Internet, links with different capacities and delays, which allows the development and the performance evaluation of P2P

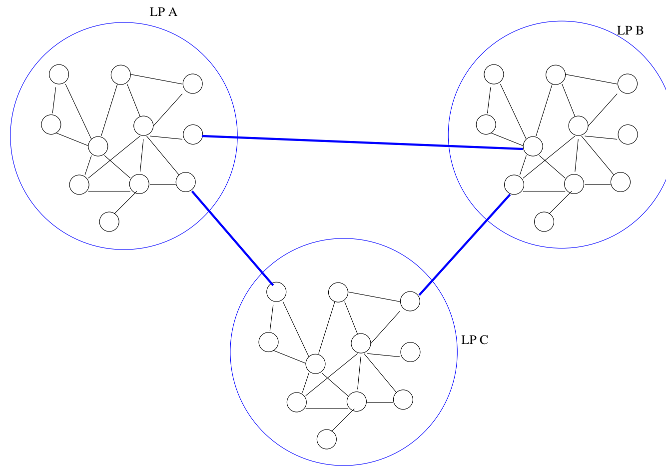


Figure 39: Example physical topology, suitable for simulation.

applications depending on the topology.

4.3 Topology specific optimizations

Exchanging high number of messages on a small-scale network is not a problem, since, according to our partitioning, all small-scale networks are completely located at a single simulator LP. Causality errors may occur when small-scale networks are exchanging messages. Normally, when a message leaves a simulator LP, it receives the current virtual time T1 of that LP as a time stamp. When it arrives at its destination, it may not be processed before the time T1 + Dl, where Dl is the minimum link delay. If T1 + Dl is greater than the time T2 of the receiving simulator LP, then everything is fine and the event may be processed, when T2 reaches T1 + Dl. A problem occurs when T1 + Dh is less than T2, where Dh is the maximal link delay. One solution would be to drop this packet, but then the system would loose packets depending on the load situation of the LPs, which is not desirable in most cases. Another solution is not to allow T2 to get higher then T1 + Dh by using synchronization (The destination LP has to wait, if its time gets too high). Consequently one can conclude, that the difference between Dm and Dh influences the frequency of the synchronizations between the two LPs - the bigger this difference, the longer the two LPs can run without synchronization (having the same virtual time jitter).

Our approach uses such a variable link delay scheme. We specify a minimum and maximum link delay Dm and Dh of a link. Messages may receive a delay that is between Dm and Dh depending on the current difference between the virtual times of the two LPs. The message delays will be minimal if the LPs are equally "virtually" loaded, so that their virtual timers advance at the same speed. The upper bound of the link delay between them would act as a buffer for small, non-constantly growing variations of the timer difference. But if the LPs are not equally loaded, the faster LP has to wait for the slower and the effect of the delay disappears.

The synchronization between two LPs may be described as follows:

- At the beginning of the simulation each of the LPs has virtual time 0 and sets its synchro-

nization deadline to the min(D2a, D2b, ...D2x), where D2a - D2x are the upper delays of all links La - Lx and are currently also the deadlines for these links.

- When a synchronization deadline is reached, the LP stops and sends its current time (via a *null message*) to the link with the minimal deadline Lx. A response should come with the current time Tx of the opposite LP. The new deadline of the link Lx is set to Tx + D2x. The new minimal deadline is calculated and the simulation is continued until it is reached.

- When a LP receives a null message, indicating that the opposite LP has reached its sync deadline, it recomputes its deadline on that link and answers with its current time. The answer can eventually be delayed according to a strategy that is not described here. The purpose of the delay of the reply is to avoid sending too many messages if the sync deadlines are increased by too small advances.

- When a LP receives a normal message (not a null message), it uses its time stamp to correct the deadline of the receiving link. When enough normal messages are sent, the need for null messages can be drastically reduced

One nice effect of this approach, is the delay aggregation across the network topology. Only simulator LPs that are direct neighbors (and are connected by a link) need to be synchronized. Thats why, the virtual timer difference between two arbitrary LPs may be as high as the aggregated maximum delay of the links along the shortest path (in terms of delay) between the LPs. The usefulness of this effect can be illustrated by looking at locality increasing activities in P2P networks. If for some reason a certain region of the network shows an increased activity, due to network outages, new node arrivals, social phenomenas, etc., the simulation load on its simulator LPs increases and their virtual timers get slower. But due to the delay aggregation effect, regions that are far away from that increased activity may continue doing their calculations without having to wait for synchronization. Figure 40 shows such a scenario. The network partition simulated by LP A has increased load due to some locally caused activity. The activity spreads to LP B, but does not cause the same load amount. LP C is not influenced by the activity. Consequently the clock of LP C will advance faster as the clock of LP B and much faster as the clock of LP A, which is possible when using the link delays. If this situation lasts, it will become necessary for LP C to wait for the other two LPs. Eventually, however, the local activity at LP A will be over before that time. Another local activity at LP C would reduce the clock differences.

4.4 Evaluation results

A parallel discrete event simulator with the described delay behavior was implemented and tested on two machines with 3 GHz CPUs and 1G Memory, connected to the same LAN segment. Each of the machines was running a single LP, the two LPs were connected with a full-duplex virtual link. Each LP was simulating 15 network nodes, which were receiving and processing packets from other nodes. Each of the 30 nodes could send packets to all other nodes, even if they were running at the other LP.

At virtual time (VT) 1.0s all network nodes received a packet. Only one type of packet with a process time of 0.01s was used for simplicity. After processing a packet, the processing node

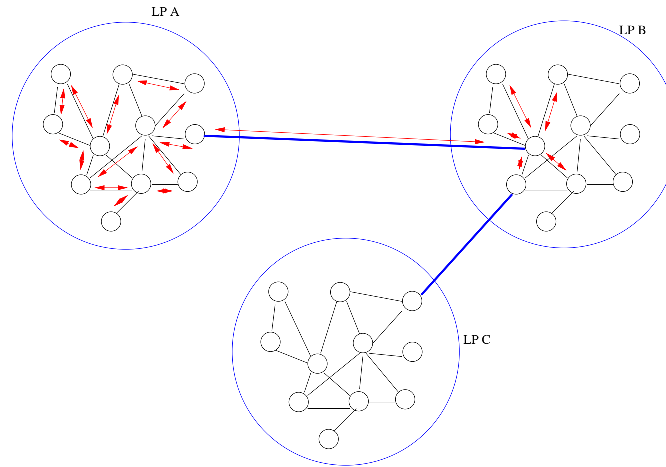


Figure 40: Example of local activity.

immediately sends the same packet to a random node between 1 and 30. One can see, that the total packet count in the system was always 30. The total packet count processed by a single LP however was a random number between 0 and 30. The number of packets currently processed on the LP determines the load of that LP and also the advance rate of its virtual timer. Conclusively, the advance rate of the timers of the two LPs was equal in average, but for short periods variations could occur. Figure 41 shows how the synchronization overhead (in terms of null messages) depends on the load between the two LPs. It also compares the overhead for delays of 0.1s and 0.001s on the virtual link between the LPs. One can see, that with a delay of 0.1s (10 times higher than the duration of a single packet processing), an increase of the null message curve occurs only in the very low regions of the command curve, which demonstrates the buffering properties of the chosen delay. A delay of 0.001s is 10 times lower than the processing time of a single packet, which causes the two LPs to synchronize 10 times between each time advance of 0.01s. This causes an enormous overhead in terms of synchronization, which is almost not dependent on the current load situation at the LP. The overhead is especially high in the beginning of the simulation (before VT 1.0) because there are no packets in the system and the only job of the two LPs is to send a null message at every 0.01s VT.

To investigate the influence of aggregated delays on the VT difference between LPs, the experiment was extended with one additional physical machine and LP, running 15 nodes, each being responsible for an additional packet. The packets were delivered by the same scheme: to a random node from 0 to 45, packet processing time was the same. Two of the three LPs (LP1 and LP3) did not have a direct connection, the packet delivery to a not directly connected LP had to be routed by the intermediate LP2 (over two 0.1 delay virtual links). Figure 42 shows the VT difference behavior between the LPs depending on the aggregated delay on all links between them. One can see, that the maximum delay difference between LP1 and LP3 is almost 0.2s, but the delay between LP1 and LP2 or between LP2 and LP3 never exceeds 0.1s, which is the desired effect of our strategy.

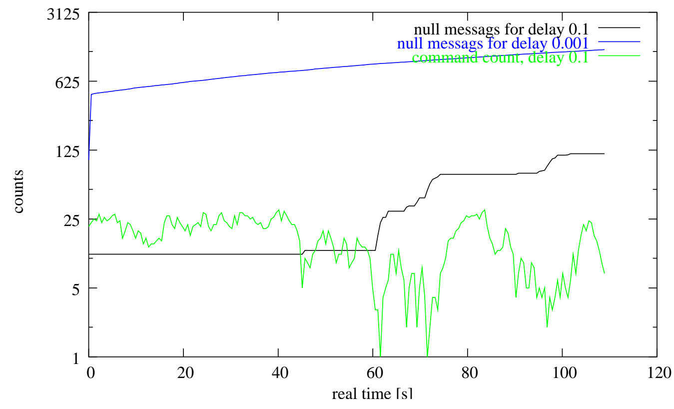


Figure 41: Null messages for LP 1 with delays 0.1s and 0.001s.

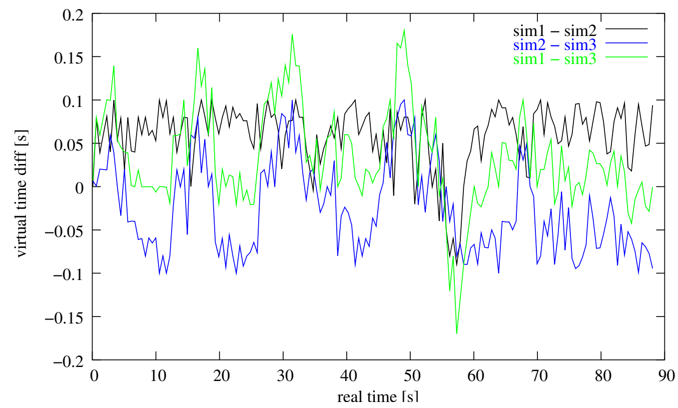


Figure 42: Delay aggregation across 3 LPs

5 Conclusions

In this technical report we investigated how large scale P2P networks can be efficiently evaluated. The algorithms and mechanisms of the P2P technology are often applied to networks and services with a demand for scalability, i.e. a large number of peers. Some approaches were presented showing the possibilites in the simulation of large scale P2P networks. We brought together the experience of different researchers of the P2P community and showed that efficient approaches for simulations can be applied with respect to compuational costs and memory consumption:

- efficient data structures,
- appropriate abstractions and models,
- clever parallel simulation techniques.

However, the different possibilities to increase simulation efficiency can only be applied for appropriate scenarios and applications. A general improvement can not be formulated, different scenarios and applications also require different simulation approaches. While one simulation might still be accurate when neglecting the physical layer, the outcome of another simulation might heavily depend on such mechanisms.

Nevertheless, this work presents some ideas, methods and algorithm, which greatly help to improve the scalability of any large scale P2P simulation. For example, we showed efficient data structures, which are able to cope with the number of events generated in a huge overlay network. In the majority of cases, finding the appropriate abstraction and models is already a big step forward. In this context, we presented examples for different levels of application in order to show how to approach a given problem. If the mere logic of a simulation is already highly optimized, a parallelization of the evaluation step sometimes does the trick. In this sense, this technical report was intended to provide insight into the real issues of large scale P2P simulation and to present approaches of how to solve them.

Acknowledgments

The authors would like to thank Prof. Phuoc Tran-Gia, Prof. Hermann de Meer, Prof. Jörg Eberspächer, and Prof. Ulrich Killat for enabling and supporting this work. Furthermore, we would like to thank Dr. Kurt Tutschku for the help in organizing joint research work and the fruitful discussions during the course of this work. A part of this work is sponsored under grant IST-50190293.

References

[1] R. Brown, "Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem," *Commun. ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

[2] K. L. Tan and L.-J. Thng, "Snoopy calendar queue," in *WSC '00: Proceedings of the 32nd conference on Winter simulation*, (San Diego, CA, USA), pp. 487–495, Society for Computer Simulation International, 2000.

[3] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 2, pp. 157–209, 1997.

[4] J. Ahn and S. Oh, "Dynamic calendar queue," in *SS '99: Proceedings of the Thirty-Second Annual Simulation Symposium*, (Washington, DC, USA), p. 20, IEEE Computer Society, 1999.

[5] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *IPTPS 2002*, (MIT Faculty Club, Cambridge, MA, USA), March 2002.

[6] H. Jin, Y. Pan, N. Xiao, and J. Sun, eds., *Grid and Cooperative Computing - GCC 2004: Third International Conference, Wuhan, China, October 21-24, 2004. Proceedings*, vol. 3251 of *Lecture Notes in Computer Science*, Springer, 2004.

[7] N. Christin, A. S. Weigend, and J. Chuang, "Content availability, pollution and poisoning in file sharing peer-to-peer networks," in *EC '05: Proceedings of the 6th ACM conference on Electronic commerce*, (New York, NY, USA), pp. 68–77, ACM Press, 2005.

[8] K. Aberer, M. Punceva, M. Hauswirth, and R. Schmidt, "Improving data access in p2p systems," *IEEE Internet Computing*, vol. 6, no. 1, pp. 58–67, 2002.

[9] H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman, "Simulation of dynamic data replication strategies in data grids."

[10] *ns-2 (The Network Simulator).* Sources and Documentation from http://www.isi.edu/nsnam/ns/.

[11] B. Cohen, "Incentives build robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer Systems*, (Berkeley, CA), June 2003.

[12] M. Izal, G. Urvoy-Keller, E. W. Biersack, P. Felber, A. A. Hamra, and L. Garcés-Erice, "Dissecting BitTorrent: Five months in a torrent's lifetime.," in *Passive and Active Measurements*, pp. 1–11, April 2004.

[13] D. Stutzbach and R. Rejaie, "Characterizing Churn in Peer-to-Peer Networks," Technical Report CIS-TR-05-03, University of Oregon, June 2005.

[14] "The pingER project," 2005. `http://www-iepm.slac.stanford.edu/pinger`.

[15] M. Castro, P. Druschel, Y. Hu, and A. Rowstron, "Exploiting network proximity in distributed hash tables," in *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, (Bertinoro, Italy), June 2002.

[16] R. Nagel, "??," Master's thesis, Munich University of Technology (TUM), Munich, Germany, 2005.

[17] J. Winick and S. Jamin, "Inet-3.0: Internet topology generator," Tech. Rep. CSE-TR-456-02, Department of EECS, University of Michigan Ann Arbor, 2002.

[18] "Brite: Boston university representative internet topology generator." `http://www.cs.bu.edu/brite/index.html`.

[19] "Cooperative association for internet data analysis (CAIDA)." `http://www.caida.org`.

[20] T. E. Ng and H. Zhang, "Towards global network positioning," in *Internet Measurement Workshop, Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement 2001*, (San Francisco, CA, US), pp. 25–29, November 2001.

[21] L. Tang and M. Crovella, "Geometric exploration of the landmark selection problem," in *Lecture Notes in Computer Science 3015, Proceedings of Passive and Active Measurement Workshop (PAM2004)*, (Juan-les-Pins, FR), pp. 63–72, April 2004.

[22] J. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965.

[23] T. Hoßfeld, A. Mäder, K. Tutschku, P. Tran-Gia, F.-U. Andersen, H. de Meer, and I. Dedinski, "Comparison of Crawling Strategies for an Optimized Mobile P2P Architecture," Tech. Rep. 356, University of Würzburg, 4 2005.

[24] L. Tang and M. Crovella, "Virtual landmarks for the internet," in *Internet Measurement Conference, Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement 2003*, (Miami Beach, FL, USA), pp. 143–152, October 2003.

[25] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, "Designing a DHT for low latency and high throughput," in *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, (San Francisco, CA, USA), March 2004.

[26] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," in *Proceedings of the ACM SIGCOMM '04 Conference*, (Portland, OR, USA), August 2004.

[27] B. Wong, A. Slivkins, and E. G. Sirer, "Meridian: a lightweight network location service without virtual coordinates," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 85–96, 2005.

[28] T. eMule Project, “eMule Source Code.” URL: `http://prdownloads.sourceforge.net/emule/`, October 2003.

[29] T. Hoßfeld, K. Leibnitz, R. Pries, K. Tutschku, P. Tran-Gia, and K. Pawlikowski, “Information Diffusion in eDonkey Filesharing Networks,” in *ATNAC 2004*, (Sydney, Australia), p. 8, 12 2004.

[30] R. Sanchez, J. Martinez, J. Romero, and R. J”arvel”a, “TCP/IP Performance over EGPRS network,” in *IEEE Vehicular Technology Conference (VTC 2002 fall)*, September 2002.

[31] R. Fujimoto, “Parallel discrete event simulation,” *Commun. ACM*, vol. 33, no. 10, pp. 30–53, 1990.

Appendix A: Example of the Market-Based Bandwidth Allocation Algorithm

The bandwidth allocation example for the MBBA algorithm in Section 3.3.2 is depicted in detail here. In this example we consider a bipartite network graph consisting of nine peers. Every peer has several logical network links in upload and download direction to other peers. The totally available capacity of each peer (for each direction) is printed beside the peer. Connections with an unassigned bandwidth are drawn as solid black lines. The bids of a peer for a connection (between an uploading and a downloading peer) are depicted by dark green numbers at the connection link nearby the peer. If the connected peers have agreed on a bandwidth the color of the connection is changed and the bandwidth is specified with the same color in the middle of the connecting line.

Figure 50 depicts a situation in which the constraints for the bid of the last connection become evident. The assignment of the bandwidth for the connection between network link 2 and 4 is done because the connection is minimal in terms of the MBBA for network link 4. If network link 8 adjusted its bid to the bids of network link 2, this would result in a lower connection speed in the next bidding round and therefore the bandwidth assignment in the next round would be 31.666 kbps instead of 36.666 kbps.

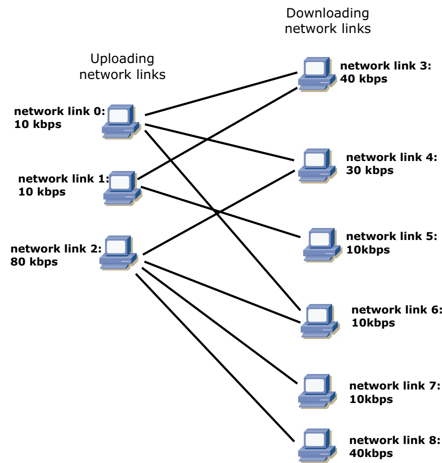


Figure 43: Considered example network for market-based bandwidth allocation algorithm

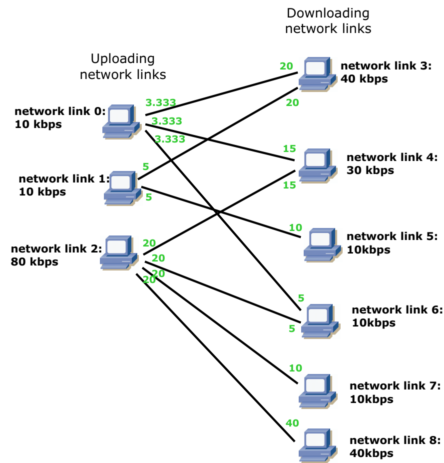
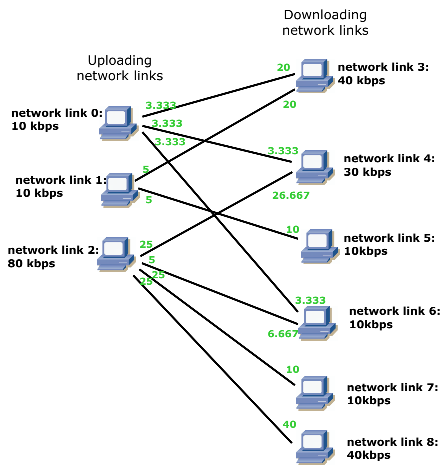


Figure 44: Initialization phase: At the beginning for all network links their capacity is divided by the number of active network connections. The initial bids are set to these values.



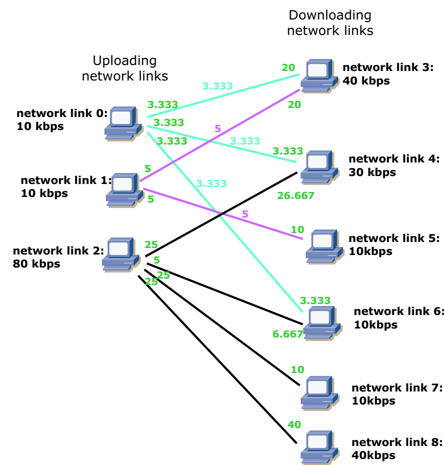
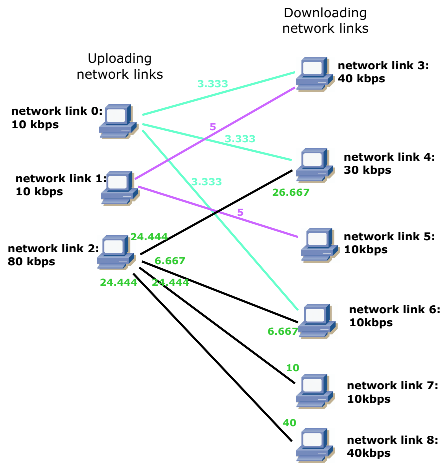


Figure 45: Bids at the first round: Network link 8 keeps the bid, although the bid of network link 2 is lower. This corresponds to the fact that his is the only and therefore last connection to be assigned for network link 8.

Figure 46: Connection speeds assigned after first bidding round: The bandwidth for all connections of network link 0 and network link 1 are assigned.



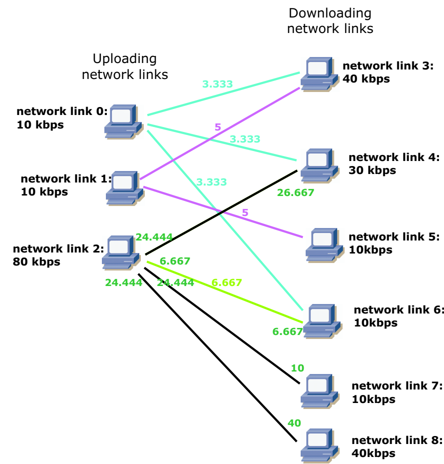


Figure 47: Bids at the second round: The network links 0, 1, 3, and 5 are no longer considered, since values to all their connections were already assigned. The MBBA is only applied to the remaining links and connections.

Figure 48: Connection speeds assigned after second bidding round: Only the connection between network link 2 and network link 6 can be assigned. There are no other links which are set to be minimal in terms of the MBBA.

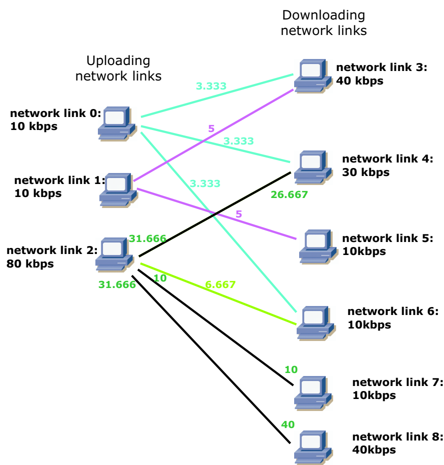


Figure 49: Bids at the third round: Only network links 2, 4, 7, and 8 have to be considered.

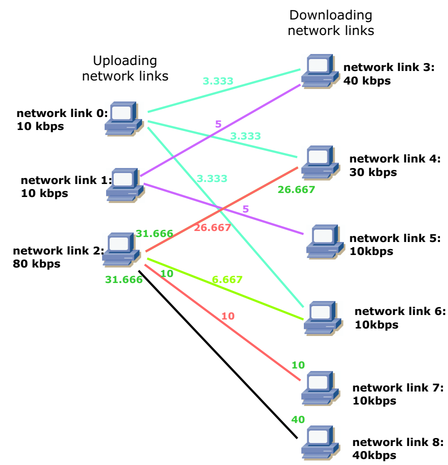


Figure 50: Connection speeds assigned after third bidding round: The connection speed between network link 2 and 4 and the connection speed between network link 2 and 7 is determined.

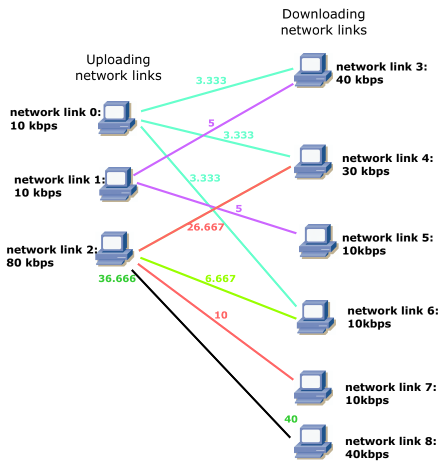


Figure 51: Bids at the fourth bidding round: Only network link 2 and network link 8 bid for their connection.

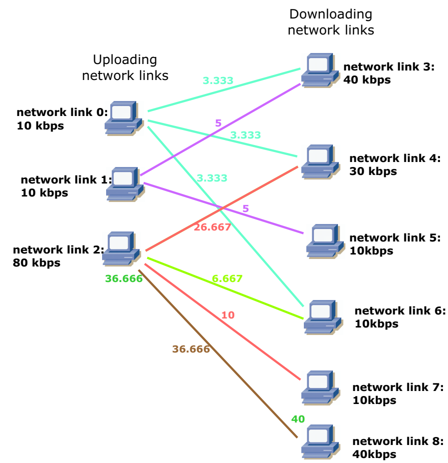


Figure 52: Final bandwidth assignment: The bandwidth of the connection between network link 2 and 8 are established with the remaining capacity of network link2.