TR-DISI-08-074: SSSim: Simple and Scalable Simulator for P2P Streaming Systems

Luca Abeni, Csaba Kiraly, Renato Lo Cigno [1]
DISI – University of Trento, Italy
{Luca.Abeni;Csaba.Kiraly;Renato.LoCigno}@unitn.it

Technical Report TR-DISI-08-074

[1]This work is supported by the European Commission through the NAPA-WINE Project (Network-Aware P2P-TV Application over Wise Network – www.napa-wine.eu), ICT Call 1 FP7-ICT-2007-1, 1.5 Networked Media, grant No. 214412

This page was intentionally left blank

TR-DISI-08-074: SSSim: Simple and Scalable Simulator for P2P Streaming Systems

author Luca Abeni, Csaba Kiraly, Renato Lo Cigno
DISI – University of Trento, Italy
{Luca.Abeni;Csaba.Kiraly;Renato.LoCigno}@unitn.it

June 19, 2009

Abstract

This technical report describes SSSim, the **S**imple and **S**calable **Sim**ulator for P2P streaming systems. SSSim can be used to evaluate the streaming performance of different chunk and peer scheduling algorithms, and is designed for performance and scalability allowing the simulation of the diffusion of a very large number of chunks over large number of peers in reasonable times. This result has been obtained by optimising the simulator for recurrent workloads and minimising the memory footprint.

1 Introduction

P2P streaming and in particular P2P support for IP-TV are becoming not only hot research topics, but also available systems and services like [Liu07, HHX$^+$03, HLL$^+$06].

The key property for the support of live streaming is the guarantee of a low distribution delay of all (or most of) the information to all peers. The property is strictly related to the overlay characteristics and the scheduling algorithms that distribute chunks to peers.

This technical report describes the SSSim simulator, designed to efficiently handle large scale P2P TV distribution systems and simulate transmissions with long TV streams. The simulator is easily extendible with new scheduling algorithms.

The technical report is organised as follows. Section 2 introduces and defines the system we consider, and the approximation we make for the formal statement of the problem. Section 3 introduces some of the scheduling algorithms we consider, both for chunk and peer selection. Section 4 describes the SS-Sim simulator, its principles and the design decisions made. Section 6 presents

[*]This work is supported by the European Commission through the NAPA-WINE Project (Network-Aware P2P-TV Application over Wise Network – www.napa-wine.eu), ICT Call 1 FP7-ICT-2007-1, 1.5 Networked Media, grant No. 214412

some simulation results showing both the scalability of SSSim as well as some properties of the studied algorithms.

2 Problem Statement

We study the scheduling procedure (chunk and peer selection) for dissemination at each peer in random, non structured overlay networks. We consider only distributed algorithms, and the goal is defining bounds for these algorithms. It is well known that the lower bound on the dissemination delay of any piece of information, given that nodes have exactly the bandwidth necessary for the streaming itself, is $\delta_{lb} = (\lceil \log_2(N) \rceil + 1)T$ where T is the transmission time[1]. It is also known [Liu07] that centralised schedulers can distribute every chunk of a stream in exactly δ_{lb}. Also, in [BMM+08] it was proved that a bound $\mathcal{O}(\log_2(N))$ holds almost surely for several *distributed* schedulers if $N \to \infty$ and $M_c \to \infty$ (M_c is the number of chunks). However, when real-time distribution systems for live events are considered an asymptotic bound $\mathcal{O}(\log_2(N))$ is not equivalent to δ_{lb}.

The contribution of this paper focuses on formally prove the existence of an entirely distributed optimal algorithm, and in finding robust, feasible schedulers that in realistic conditions perform within a reasonable bound of the optimal one. This contribution is the necessary starting point (a reference optimum) for further research focusing on heterogeneous systems, on the interaction of the overlay with the underlying IP network and on all those 'impairments' that forbids finding closed-form formal solutions to problems in real networking scenarios.

2.1 System Description

The system considered is an overlay network of peers connected with a general mesh topology. The total number of peers is N. Each peer is connected to NN other peers[2] which constitute its *neighbourhood.* A special case is when $NN = N - 1$, which define a fully connected mesh network. We consider the presence of one more "special peer" in the system that is the source of the video. The source never receives chunks, so its links are logically mono-directional and it is not part of any neighbourhood, i.e., its mono directional links are additional to the others. Fig. 1 reports two sample topologies.

The source is a video server that distributes a (possibly live) video or TV program. The video is divided in M_c *chunks* of equal duration emitted periodically by the source. All peers have unit bandwidth (i.e., they can transmit a chunk in exactly the inter-chunk generation time) on the uplink and no limitations on the downlink. We do not consider churn (i.e., the process of joining

[1]The bound comes from the fact that each node can transmit the chunk only after receiving it, and the number of nodes owning the chunk at most doubles every T.

[2]For the sake of simplicity we restrict discussion to n-regular topologies: random graphs with symmetric connectivity and n links per node.

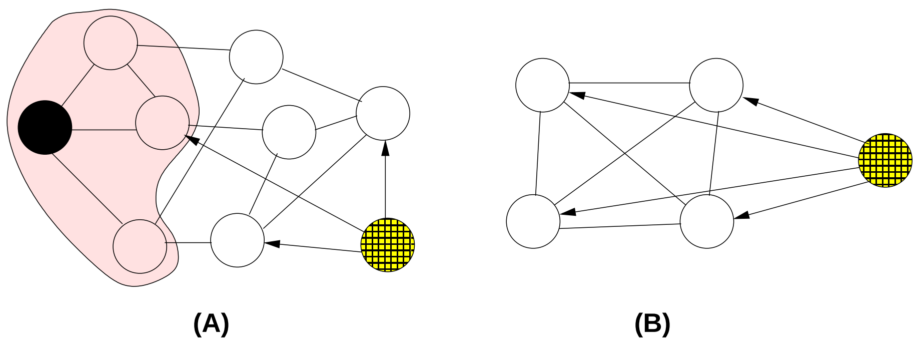


Figure 1: (A) – General mesh topology with $N = 8$ and $NN = 3$; the shaded (pink) area is the neighbourhood of the black node; the source is the checkered (yellow) node;
(B) – Full mesh with $N = 4$

and leaving the overlay by peers) and we focus, as main performance parameter, on the diffusion time of chunks, which is the delay (after their emission from the source) with which chunks are received by all peers. Formally, if r_i is the emission time of chunk C_i, then its diffusion time f_i is the delay $\delta = t - r_i$ such that all N peers have received C_i.

The scheduling procedure is entirely distributed, but each peer has a perfect knowledge of the status of its neighbours. This means that i) no global ordering of peers is required; ii) the system is not structured; iii) schedulers' decisions are independent one another; vi) peers know exactly the subset of chunks already received or being received by all neighbours; and v) signalling delay is negligible.

The first scheduling decision, which also affects the protocol supporting the streaming, is whether a peer *pushes* information to other peers or if it *pulls* it from other peers . . . or a mix of the two policies. Sometimes in the literature it is stated that pushing information is a behaviour typical of structured systems, and pull methods are more adapt for non-structured overlays. Recent papers like [CdSLMM08, BMM$^+$08] instead used push schedulers on non-structured meshes. Indeed, the choice of whether it is better to push or pull information is not related to the structure (or the lack of it) of the system, but to the bandwidth bottleneck, which can create conflicts in scheduling decisions.

Push-based systems are suitable for systems where the bandwidth bottleneck is the uplink, because this guarantees a priori that only one chunk will be scheduled for transmission on the uplink, and that scheduling conflicts arising from the distributed nature of the scheduling will insist on the downlink of other peers, which has more resources and can accept multiple downloads.

If the situation were reversed (uncommon in networks dominated by ADSL access links, but technically possible), then pull-based schedulers would solve a priori the conflict on the downlink, and more bandwidth-endowed uplinks would accommodate scheduling conflicts. Interestingly, a scenario with symmetric up-

and downlink capacities does not offer an easy logical choice on whether pushing or pulling information is the best choice.

Given the networking scenario depicted we obviously consider push-based schedulers, but we claim that reversing the bottleneck hypothesis, pull-based schedulers which are dual to those we prove optimal in the sequel can be easily derived.

2.2 Formal Notation and Definitions

A system is composed by a set $\mathcal{S} = \{P_1, \dots P_N\}$ of N peers P_i, plus a special node called *source*. Each peer P_i receives chunks C_j from other peers, and send them out to other peers at a rate $s(P_i)$. The source sends out chunks with a rate $s(source)$. The set of chunks already received by P_i at time t is indicated as $\mathcal{C}(P_i, t)$.

The source, not included in $\mathcal{S}$, generates chunks in order, at a fixed rate λ (C_j is generated by the source at time $r_j = \frac{1}{\lambda}j$). We normalise the system w.r.t. λ, so that $r_j = j$. Also, we set $\forall i, s(P_i) = s(source) = \lambda = 1$, which is the limit case to sustain streaming.

If $\mathcal{D}_j(t - r_j)$ be the set of nodes owning chunk C_j at time t, the worst case diffusion time f_j of chunk C_j is defined as the time needed by C_j to be distributed to every peer: $f_j = \min\{\delta : \mathcal{D}_j(\delta)) = \mathcal{S}\}$. According to this definition, a generic peer P_i will receive chunk C_j at time t with $r_j + 1 \le t \le r_j + f_j$, and when considering an unstructured algorithm t will be randomly distributed inside such interval. Hence, in an unstructured system P_i is *guaranteed* to receive C_j at most at time $r_j + f_j$. To correctly reproduce the whole media stream, a peer must buffer at least $F = \max_{1 \le j \le M_c} f_j$ chunks before starting to play. For this reason, the worst case diffusion time F is an important performance metric for P2P streaming systems, and this paper will focus on it.

When $\forall i, s(P_i) = \lambda = 1$, at time t the source sends a chunk C_j (with $r_j = t$) to a peer and every peer P_i sends a chunk $C_h \in \mathcal{C}(P_i, t)$ to a peer P_k. All these chunks will be received at time $t + 1$.

As discussed earlier, the minimum possible diffusion time f_j for chunk C_j is $\lceil log_2(N) \rceil + 1$. Chunk diffusion is said to be optimal if $\forall j, f_j = \lceil log_2(N) \rceil + 1 = F$.

3 Scheduling Peers and Chunks

In a push-based P2P systems, when a peer P_i sends a chunk, it is responsible for selecting the chunk to be sent and the destination peer. More formally, every time that P_i sends a chunk $C_j \in \mathcal{C}(P_i, t)$ to a peer P_k, two scheduling decisions are taken:

- The chunk C_j to be sent is selected by a *chunk scheduler*;
- The destination peer P_k is selected by a *peer scheduler*.

This paper considers algorithms which first select the chunk C_j, and then select a target peer P_k which needs C_j.

Some well known chunk scheduling algorithms are *Latest Blind Chunk*, *Latest Useful Chunk*, and *Random Chunk* (again, blind or useful). The Latest Blind Chunk algorithm schedules at time t the latest chunk:

$$C_j \in \mathcal{C}(P_i, t) : \forall C_h \in \mathcal{C}(P_i, t), r_j \geq r_h \tag{1}$$

C_j is scheduled even if all the other peers already have it. The Latest Useful Chunk (LUc) algorithm modifies the chunk scheduler to select a chunk that is needed by at least one peer. Formally:

$$C_j \in \mathcal{C}'(P_i, t) : \forall C_h \in \mathcal{C}'(P_i, t), r_j \geq r_h \tag{2}$$

where $\mathcal{C}'(P_i, t)$ is a subset of $\mathcal{C}(P_i, t)$ containing only chunks that have not already been received (or are not currently being received) by other peers. The Random Chunk algorithms select a random chunk in $\mathcal{C}(P_i, t)$ (Random Blind Chunk) or in $\mathcal{C}'(P_i, t)$ (Random Useful Chunk – RUc).

Once the chunk C_j to be sent has been selected, the peer scheduling algorithms selects a peer P_k which needs C_j. If the LUc or RUc algorithm is used for chunk scheduling, P_k can always be found.

The most commonly used peer scheduling algorithm is Random Peer, which randomly selects a peer which needs C_j. In theory, the chunk scheduling algorithm can select $P_k \in \mathcal{S}$, but in practice peer P_i will only know a subset of all the other peers, and will select P_k from a subset of $\mathcal{S}$ called *neighbourhood*. The neighbourhood of peer P_i is indicated as $\mathcal{N}_i$.

The case in which $\forall i, \mathcal{N}_i = \mathcal{S} - P_i$ is special, and corresponds to a fully connected graph.

3.1 Optimal Peer Scheduling

Random peer selection prevents achieving optimality, because the selected peer might be unable to further distribute the chunk. The rationale behind optimal peer selection should be the following: the selected destination peer should be able to immediately take on the role of redistributing the chunk.

We define the "Earliest-Latest" peer scheduler (ELp) as follows: ELp selects as target a peer P_l that needs C_h and owns the latest chunk C_k with the earliest generation time r_k:

$$C_h \notin \mathcal{C}(P_l, t) \wedge \forall P_j \in \mathcal{S}, L(P_l, t) <= L(P_j, t) \tag{3}$$

where $L(P_i, t) = \max_k \{r_k : C_k \in \mathcal{C}(P_i, t)\}$ is the latest chunk owned by or in arrival to P_i at time t. If at time t P_i has not received any chunk yet, $L(P_i, t) = 0$. If more peers exist that satisfy (3) one is chosen at random.

3.2 Optimal Chunk Scheduling

The LUc/ELp algorithm is optimal when used on a full mesh. Unfortunately, when the neighbourhood size is reduced the worst-case performance of the

LUc/ELp algorithm are badly affected (see Sect. 6) because any time that limited knowledge of the neighbourhood make a later chunk arrive to a peer before an earlier one, the diffusion of this latter is stopped.

To solve this problem, an algorithm that embeds the number of replicas of a chunk is needed. We define a *deadline-based* (Dl) scheduling algorithm that works as follows:

1. Each chunk C_k is assigned a *scheduling deadline* d_k, which is initialised to $d_k = r_k + 2$ when the source sends C_k at time r_k;
2. When peer P_i has to send a chunk at time t, the Dl scheduler selects the useful chunk C_k having the minimum scheduling deadline:
$$C_k : \forall C_h \in \mathcal{C}'(P_i, t), d_k \leq d_h; \tag{4}$$
3. Before sending C_k its scheduling deadline is incremented by 2 time units: $d_k = d_k + 2$ (both P_i and the destination peer will see C_k with its new scheduling deadline, while chunk instances present in other peers are obviously not affected).

Note that the scheduling strategy based on selecting the chunk with a minimum deadline is known in literature as Earliest Deadline First (EDF), and is mentioned as "Deadline Driven Scheduling" in a seminal paper by Liu and Layland [LL73], but to the best of our knowledge, it has never been applied with dynamic deadlines in distributed systems.

Observation 1 *The scheduling deadline d_k of a chunk instance C_k of peer P_i is equal to $r_k + 2d$, where d is the number of times that C_k has been selected by the Dl schedulers along the path taken by the chunk till P_i.*

4 Simulating a P2P Streaming System

While the most important properties of a chunk/peer scheduling algorithm have to be formally proved, simulating the chunk diffusion is often very important to understand the behaviour of an algorithm, to estimate its performance, and to develop new algorithms.

Being the scheduling algorithms the core and engine of P2P systems, a simulator focusing specifically on scheduling, and abstracting to a higher level of approximation the remaining parts of the system is needed. The most important requirements for a such simulator are *simplicity* (so that implementing new features to evaluate particular aspects of a scheduling algorithm is straightforward) and the possibility to *easily implement new scheduling algorithms* in a flexible way.

Moreover, when developing a new scheduling algorithm it is important to evaluate its performance in many different situations, and to ensure that the algorithm works well for a large number of peers. For a streaming algorithm, long-term stability (the fact that the diffusion delay of a chunk does not increase with the chunk number) is also important. Hence, *the simulator must be able to simulate the diffusion of a large number of chunks over a large number of peers.* This requirement is very important, and has two consequences:

1. the amount of memory used by the simulator should not grow too much (in particular, it cannot be proportional to both the number of peers N and the chunk number M_c)
2. **performance matters**.

The constraint on the amount of memory used by the simulator is needed for allowing it to properly scale (for example, if the amount of required memory is proportional to both N and M_c and that a pointer and an integer are needed for every peer-chunk pair, then simulating the distribution of 20000 chunks over 10000 peers requires at least $1.5GB$ of memory). The simulator's performance are also important in order to be able to finish a simulation in a reasonable time.

Summing up, the requirements for the simulator are:

- Flexibility
- Performance
- Simplicity
- Scalability
- Possibility to easily add/implement new scheduling algorithms

As an output, the simulator should be able to produce a trace of the chunk diffusion (when used with a small number of peers/chunks, this feature is useful to check the correctness of new algorithms, and to understand their dynamics and chunk diffusion patterns) and some statistics about diffusion times (used with large numbers of peers/chunks to evaluate the algorithms' performance).

Finally, the simulator should be able to reproduce the chunk diffusion on full meshes (useful for testing the optimality of a scheduling algorithm) or on various kinds of overlays with different properties, both random as n-regular graphs or Watts-Strogats topologies, or mimicking topologies actually built by distributed algorithms. For this latter reason the possibility of reading topologies generated by external tools in standard notation like dot[3]

As most of the existing simulators do not allow to easily implement new scheduling algorithms, and we are not aware of any simulator providing the requested features in a scalable and efficient way (providing reasonable performance) a new simulator has been developed from scratch: SSSim, the **S**imple and **S**calable **Sim**ulator.

SSSim has been developed for performance and scalability, and most of the design decisions have been taken based on these two main requirements (if

[3]See the URL: http://www.graphviz.org/ for additional information on dot and graph visualisation.

scalability and performance are not important requirements, then there are probably better simulators than SSSim around).

The simulator has been released under the GPL[4] and is available at `http://imedia.disi.unitn.it/SSSim`.

4.1 General Structure

The main reason for developing SSSim was to perform optimality tests on new scheduling algorithms such as LUc/ELp and Dl/ELp [AKC09], and optimality is defined when all the nodes have the same output bandwidth (see Section 2.2). Hence, the simulator is optimised for this kind of simulations: at time t, the source generates a new chunk, and every node P_i starts sending a chunk to a target node P_j. All these chunks will be received at time $t+1$, before starting the new send cycle.

As a result, SSSim has been developed as a *discrete-time simulator* (opposed to an *event driven simulator*, that would be much less efficient in this case, requiring an event queue and all the related overhead). However, the simulator has been designed to be modular and flexible (when this does not affect performance), so most of its modules can be re-used in event driven simulations.

The simulator is organised in some software modules, interfaced using a clean and well defined API:

- Overlay generation
- Main loop
- Scheduler
- Statistics

4.2 Overlay Generation

Each peer P_i is identified by a structure of type `struct peer`, which contains a pointer `neighbour` an array of pointers to the neighbours. Such an array is filled during the simulation startup time (but can be modified during the simulation to implement churn and dynamic overlays by the *overlay generation module*. Different kinds of overlays can be used, ranging from a full mesh to a pipeline and passing through various kinds of random graphs.

The full mesh case is special, because it would require to allocate an array of $N-1$ elements for every peer P_i (and to fill it with with $\mathcal{S} - \{P_i\}$), consuming a large amount of memory (at least $4N(N-1)$ bytes, depending on the size of a pointer) which is not acceptable for scalable simulations. For this reason, SSSim provides a special graph generator which sets the `neighbourhood` pointer for all the peers to the same array, containing a list of all the peers. In this way each peer P_i will see itself in its neighbourhood (but properly written

[4]http://www.fsf.org/licensing/licenses/gpl.html

schedulers can easily cope with this without any performance loss), but the memory requirements goes down to `sizeof(void *)`N.

Another special graph generator in the *DOT graph importer*, which generates the overlay by reading the graph description (in the standard DOT language) from a file. This feature allows SSSim to interoperate with other programs that generate graphs using more complex and specialised algorithms: for example, it is possible to import an overlay generated by PeerSim [] using the newscast algorithm [VJVS03].

4.3 Scheduling

A scheduler is described by a `schedule()` function, with prototype

```
void schedule(struct peer *p, int t,
              int *chunk, struct peer **target)
```

where `p` is a pointer to the peer which has to send a chunk, `t` is the current time, and the pair (P_i, C_j) selected by the scheduler (target peer and sent chunk) are returned in `chunk` and `target`. This function is completely generic, but some specialised schedulers `schedule_c_p()` (select the chunk first and the target peer after) and `schedule_p_c()` (select the target first) are provided.

The `schedule_c_p()` and `schedule_p_c()` then call a `peer_sched()` function (that can be defined to ELp, RUp, etc..) and a `chunk_sched()` function (that can be defined to LUc, Dl, etc...).

In this way, it is possible to easily modify the chunk or peer scheduler (and, if needed, it is also possible to use a scheduler that selects chunk and peer at the same time). Other helper functions (for utility-based chunk or peer scheduling, random scheduling, etc...) are also provided for convenience; in most of the cases, a new scheduler can be easily obtained by combining existing functions, but if needed the `schedule()` function can be completely replaced with a new version rewritten from scratch.

For performance reasons, schedulers can be configured at compile-time, but not at run-time (in this way, the cost of indirect function calls and/or switches and conditional branches can be avoided). As a consequence, schedulers are activated and configured using preprocessor `#define` directives.

4.4 The Main Loop

After initialising the overlay, the simulator sets $t = 0$ and starts running the main loop that:

1. Receives "on the fly" chunks (that have been sent at time $t - 1$)

2. Generates a new chunk in the source, invokes the peer scheduler to select a target peer, and sends the chunk (setting it as "on the fly")

3. For each peer, invokes the `schedule()` function and sends out a chunk

4. Increases t, and returns to step 1

Every time that a chunk is received by a peer, it is added to an array of received chunks in the peer structure. However, while this approach can simplify the development of the simulator and of new schedulers (and makes statistics gathering very easy), it requires an amount of memory proportional to NM_c, penalising the simulator's scalability. For this reason, SSSim can be configured to use a reduced chunk buffer per peer; in this case, if B is the chunk buffer size then the amount of memory required for a simulation is proportional to NB. Buffer size handling has been made optional because in case of large buffers (near to M_c) handling the buffer size is less efficient than storing all the received chunks.

Note that the buffer size B imposes a maximum playout delay (the time after which a chunk is discarded), but if the buffer is not properly handled the relation between B and the playout delay is not immediately clear. The chunks buffer management can introduce a significant overhead in the simulation. To reduce such an overhead, SSSim inserts the chunks in the buffer according to their generation times, so that chunk C_i generated at time r_i is inserted in position $r_i \% B$ of the buffer. If $B = 2^k$, then $r_i \% B$ can be computed as $r_i \& (B - 1)$, reducing buffer management overhead significantly.

Note that the buffer size B imposes a maximum playout delay (the time after which a chunk is discarded). However, the exact relation between maximum playout delay and B depends on the algorithm used to handle the buffer. The buffer management technique presented above has the advantage that the maximum playout delay is exactly equal to B.

5 Simulator Performance

Since SSSim has been designed for performance, the time needed to finish a simulation (in different situations) has been evaluated by repeating a large set of runs on an Intel(R) Pentium(R) D CPU running at $3.40GHz$ with $2GB$ of RAM. All the experiments have been ran simulating a P2P system in which all nodes have the same output bandwidth (equal to the source rate) and have very large download bandwidth. A pretty standard scheduling algorithm (the so called latest useful chunk / random useful peer or LUc/RUp algorithm) has been used for diffusing the chunks.

5.1 Comparison with a Traditional Simulator

In a first set of experiments, SSSim performance have been compared with the performance of P2PTVSim (as an example of a more traditional, event-based simulator), in order to show the improvements due to SSSim's design and optimisations. The results are reported in Table 1.

In a first experiment, the diffusion of 3000 chunks over 1000 peers has been simulated assuming that the peers are connected by a full mesh and have a buffer size $B = 32$. From the table it is possible to notice that SSSim needed less than 1/10 of P2PTVSim's time to finish the simulation, and consumed about

N	M_c	NN	B	Time SSSim	Mem SSSim	Time P2PTVSim	Mem P2PTVSim
1000	3000	999	32	$72s$	$2.945MB$	$835s$	$23.473MB$
2000	3000	999	32	$350s$	$3.9MB$	$5609s$	$82.461MB$
3000	3000	999	32	$993s$	$4.957MB$	$17355s$	$179.723MB$
1000	3000	999	3000	$371s$	$101.801MB$	$80282s$	$114MB$
10000	10000	20	32	$355s$	$12.953MB$	$2138s$	$21.148MB$

Table 1: Comparisons between SSSim and a non-optimised simulator (P2PTVSim).

1/10 of the memory. In the following experiments, the number of peers N has been increased, showing that SSSim is able to keep under control the amount of memory and execution time needed to finish a simulation (in particular, the maximum amount of memory needed by SSSim seems to scale pretty well). These first results indicate that the optimisations introduced in SSSim to speed up the simulations on fully connected overlays are effective.

Next, the performance of the two simulators have been compared by increasing the playout delay, to test the effectiveness of the optimisations introduced in SSSim chunks buffer handling[5]. To ensure that no chunk is lost, the chunks buffer size has been increased to 3000[6] (the number of chunks), while still maintaining $N = 1000$ (to keep P2PTVSim's simulations times under control). The results in the table show that while the maximum amount of memory needed by the simulators is comparable (because most of the memory is used in the chunks buffer), SSSim is able to finish the simulation in a much shorter time. Since the chunks buffer handling mechanism can introduce some overhead (and waste some memory), when running simulations with "infinite" chunks buffer size it can be useful to disable such a mechanism. SSSim allows to compile the chunks buffer handling code out of the simulator. With this optimisation enabled, the simulation finished in $238s$ and required only $93.461MB$ of memory.

All the simulations mentioned above focused on system configurations for which SSSim has been heavily optimised (full mesh, large buffer sizes, ...). Hence, some additional simulations with reduced neighbourhood size and small playout delays have been run. Again, the results indicate that SSSim's optimisations are effective, allowing it to consume less memory than a traditional simulator and to finish the simulations in a shorter time (in these cases, the differences between SSSim's performance and P2PTVSim's ones are smaller, because SSSim has not been specifically optimised for this setup). Increasing the number of chunks, the time scaled about linearly for both the simulators (for example, 30000 chunks were diffused in $1070s$ by SSSim and in $6625s$ by P2PTVSim), and the amount of needed memory did not increase too much).

[5]Note that we are using SSSim as a testbed for these optimisations, that could also be ported to other simulators

[6]Note that in SSSim the chunks buffer size must be a power of 2, so a buffer size of 4096 has actually been used

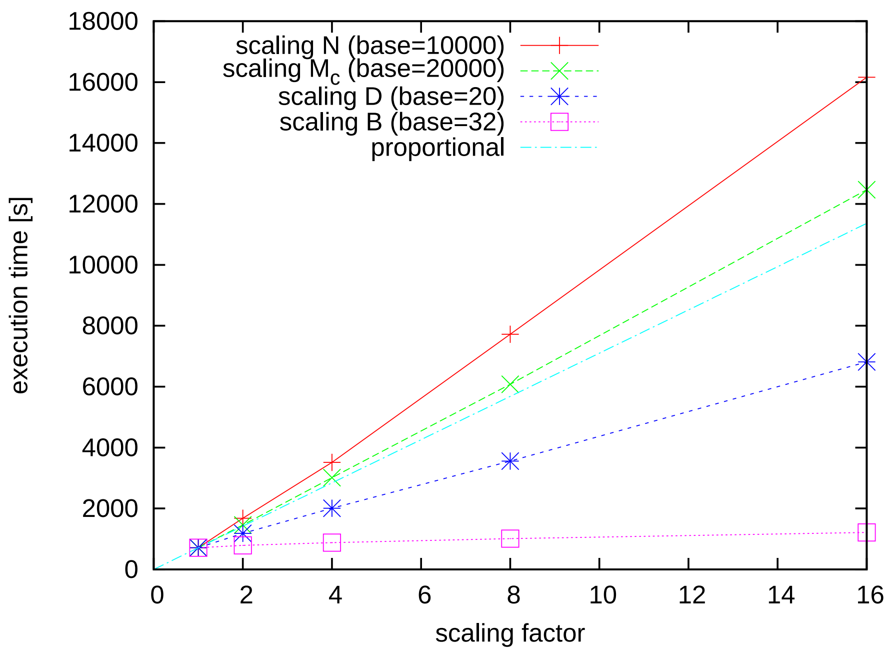


Figure 2: CPU time as a function of N, M_c, NN and B.

5.2 SSSim Scalability

In a following set of experiments, we examined SSSim's scalability, as a function of four parameters: N, M_c, NN and B. Figures 2 and 3 show CPU time and memory consumption as a function of these four parameters. Our reference parameter set is N=10000, M_c=20000, NN=20 and B = 32, and we scale one parameter at a time. The first curve for example represents simulations varying N from 10000 to 80000 while keeping all other parameters at their reference value.

The figure shows that the execution time is almost proportional to M_c which is the best we could expect since each chunk is diffused individually. With a fixed neighbourhood size, CPU time scales almost proportionally also in N, although a slight extra time is needed due to the longer diffusion time of chunks in the system. Again, we couldn't expect better scaling. Execution time scales better with NN, increasing with a small gradient. Increased computation time is due to the use of latest useful chunk selection, which should verify all neighbour's state. Scaling would be even better with different peer scheduling algorithms (such as blind ones). Finally, scaling in B is sub-linear with a very small gradient, which shows the efficiency of the implemented buffer management.

Memory usage scales almost proportionally to both N and B, which shows that the main user of memory is the chunk buffers itself. Other parameters influence memory usage only marginally. More specifically, memory usage does

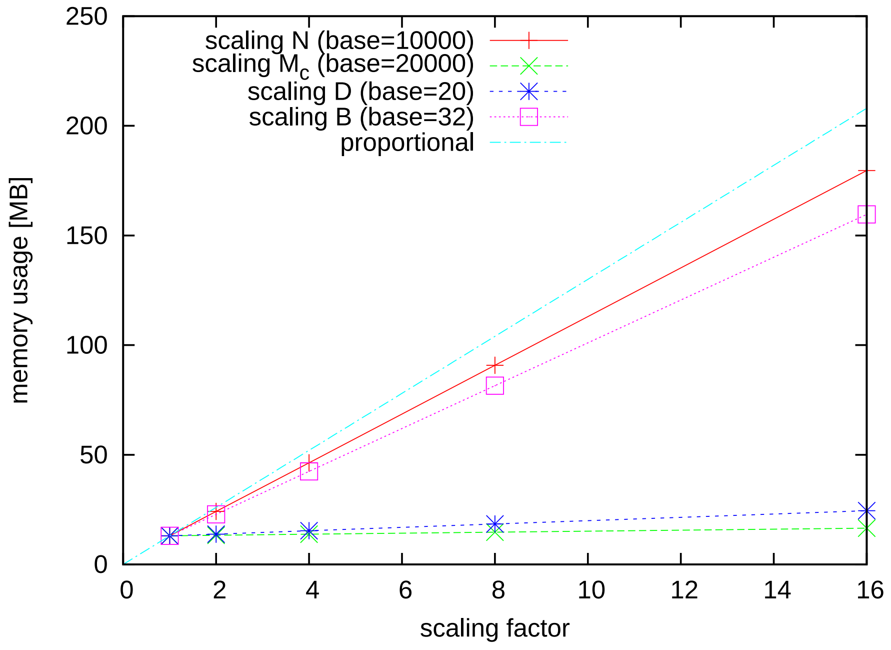


Figure 3: Memory consumption as a function of N, M_c, NN and B.

not depend on M_c, which property is very important for being able to simulate streaming systems. Memory usage increase with NN is also almost negligible.

To understand whether these scaling properties hold for a larger set of parameters, we have also run simulations varying both N and M_c. Figures 4 and 5 show both CPU time and memory usage, confirming good scaling characteristics in both parameters.

6 Some Simulation Examples

This section presents some examples of simulative results obtained by using SSSim. In particular, it shows how SSSim can be used to compare the streaming performance of different scheduling algorithms, focusing on LUc/ELp and Dl/ELp. Although such algorithms have been proved to provide optimal performance in the case of a fully connected graph ($\forall i, \mathcal{N}_i = \mathcal{S} - P_i$), their performance in more realistic situations can be analysed only through simulations.

6.1 Simulating P2P Streaming and Measuring Performance

The behaviour of the scheduling algorithms introduced in Section 3 (given as LUc, RUc, or Dl, combined with either ELp or with RUp) is analysed through simulations performed with SSSim. As defined in Section 2.1, an overlay of

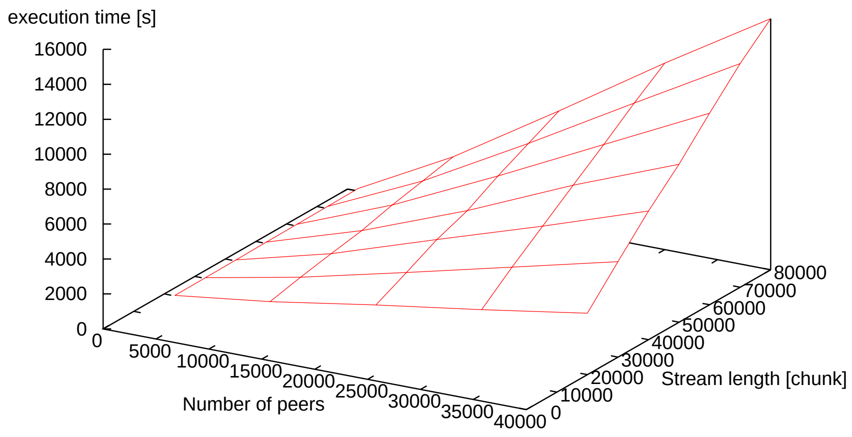


Figure 4: CPU time scaling both N and M_c ($NN = 20$ and $B = 32$).

Npeers is set up, each having unit upload and infinite download bandwidth. The source distributes $M_c$chunks.

As explained in Section 2 the performance metric considered in this paper is the worst case diffusion time F, and (as stated in Section 3), a scheduling algorithm is optimal iff $F = \lceil log_2(N) \rceil + 1$.

Figure 6 shows the worst case diffusion time of the six considered algorithms as a function of the number of peers N. Each point was obtained running the simulation 10 times, respective mean and its 90% confidence interval (due to random choices in the selection algorithms) are shown. The figure shows LUc/ELp and Dl/ELp to achieve optimal performance, significantly outperforming the other algorithms.

6.2 Restricting the Overlay

As introduced in Section 2, in realistic situations a restricted overlay is used instead of a fully connected graph. Such a restricted overlays is modelled assuming bidirectional relations and a pre-defined number ($NN = ||\mathcal{N}_i||$) of neighbour nodes. The resulting graph is modelled as a random NN-regular graph. In following simulations, algorithms are evaluated on 10 samples of the random NN-regular graph.

Figure 7 shows performance of different streaming algorithms as a function of NN and shows how the LUc/ELp algorithm (which is optimal on a full mesh) is highly sensitive to neighborhood restrictions and performs badly when $NN < N - 1$. Dl/ELp, on the other hand, works better than all the other algorithms and is able to achieve values of the worst case diffusion time f_i near to the optimum (which in this case is 11).

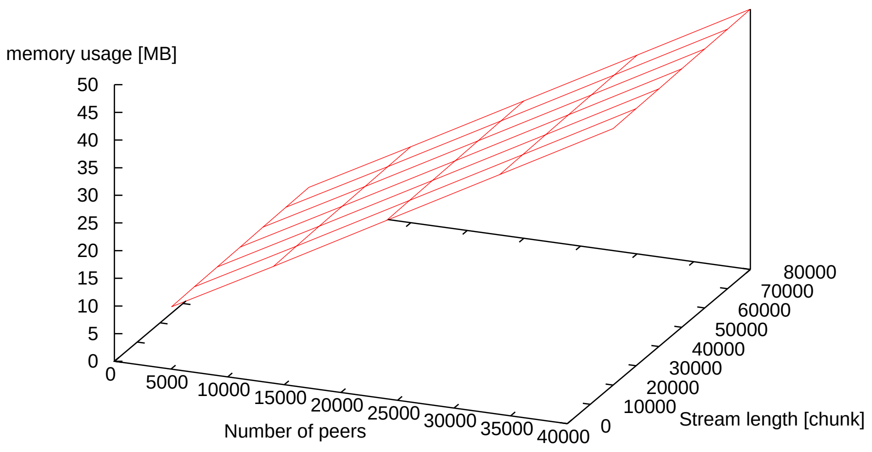


Figure 5: Memory consumption scaling both N and M_c ($NN = 20$ and $B = 32$).

After verifying that Dl/ELp seems to perform reasonably for small NN values, some additional simulations have been ran to check if the various scheduling algorithms are suitable for streaming. This has been verified by increasing the number of chunks and checking if f_i depends on the number of chunks. The result of such simulations seems to indicate that if $NN \le \log_2(N)$ then f_i always increases with i. However, when $NN > \log_2(N)$ Dl/ELp is often able to distribute the chunks so that f_i does not increase with i. This clearly depends on the graph). For example, Figure 8 shows how the number of chunks affects f_i for $NN = 8$ and $NN = 11$ (note that $N = 1000 \Rightarrow \log_2(N) = 9.9658$).

6.3 Limiting the Chunk Buffer Size

From the last simulations, it can be seen that in many situations f_i increases with i (the performance of the algorithm dependends on the stream length), hence the distribution mechanism results to be unstable in a streaming context. The only solution to this problem is to define a playout delay D, and to discard chunks C_j at time $r_j + D$. This causes some chunk loss (for chunks C_j that would have $f_i > D$), but can make the distribution system stable again. Moreover, the playout delay D can be used to dimension the chunk buffers in the peers (in particular, each peer needs to buffer at most D chunks) [7].

First of all, it has been verified that when fixing a playout delay D the performance of an algorithm become independent on the number of chunks.

[7]properly implementing such a *chunks buffer size* in the P2P simulator can enable some optimisations (reducing the memory pressure, and the computations needed by the chunk scheduler) which allow to perform simulate larger task sets. As a result, the following simulations will be performed with $N = 10000$.

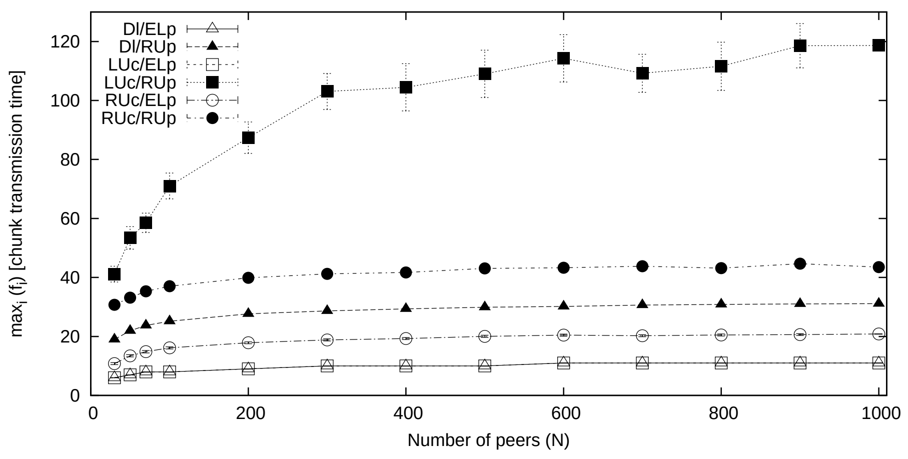


Figure 6: Full mesh overlay; maximum chunk diffusion delay as a funcion of the number of peers; 200 chunks

This has been done by simulating the diffusion of M_c chunks over 10000 peers and varying M_c. Since some chunks can be lost, the performance should be evaluated based on both chunk loss ratio and the maximum delay[8];

Figure 9 plots the results of these simulations, showing that the performance do not depend on the number of chunks (as expected, both LUc/ELp and Dl/ELp achieve the optimal value for f_i with 0 lost chunks. Hence, the introduction of a playout delay is not affecting these two algorithms on a full mesh). Based on these results, all the following simulations will be performed using only 10000 chunks.

Finally, Figure 10 plots the chunk loss ratio for the various algorithms as a function of the neighbourhood size. Note that for $NN > 14$ the chunk loss ratio for Dl/ELp is 0, showing that it is possible to dimension the chunk buffer size so that it does not affect the algorithm's performance (to the authors' best knowledge, this is not possible for the other algorithms). The worst case diffusion time f_i is not plot because it is always equal to 32 for all the algorithms but Dl/ELp.

References

[] Peersim home page. `http://peersim.sourceforge.net`.

[8]with this double metric, an algorithm A can be said to be better than another one B only if both B's chunk loss and its maximum delay is lower than those of A (TODO: MENTION PARETO?)

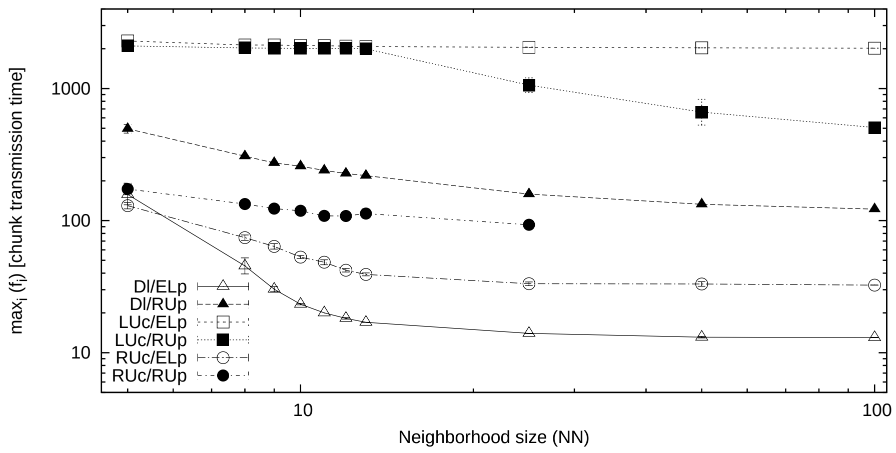


Figure 7: Worst case diffusion time as a funcion of the neighbourhood size; 2000 chunks.

[AKC09] Luca Abeni, Csaba Kiraly, and Renato Lo Cigno. On the optimal scheduling of streaming applications in unstructured meshes. In *IFIP Networking*, 2009.

[BMM+08] Thomas Bonald, Laurent Massoulié, Fabien Mathieu, Diego Perino, and Andrew Twigg. Epidemic live streaming: optimal performance trade-offs. In Zhen Liu, Vishal Misra, and Prashant J. Shenoy, editors, *SIGMETRICS*, pages 325–336, Annapolis, Maryland, USA, June 2008. ACM.

[CdSLMM08] A.P. Couto da Silva, E. Leonardi, M. Mellia, and M. Meo. A bandwidth-aware scheduling strategy for p2p-tv systems. In *Proceedings of the 8th International Conference on Peer-to-Peer Computing 2008 (P2P'08)*, Aachen, September 2008.

[HHX+03] Mohamed Hefeeda, Ahsan Habib, Dongyan Xu, Bharat Bhargava, and Boyan Botev. Collectcast: A peer-to-peer service for media streaming. *ACM Multimedia 2003*, 11:68–81, 2003.

[HLL+06] X. Hei, C. Liang, J. Liang, Y. Liu, and K. W. Ross. Insights into pplive: A measurement study of a large-scale p2p iptv system. In *Proceedings of the Workshop on Internet Protocol TV (IPTV) services over World Wide Web in conjunction with WWW2006*, 2006.

[Liu07] Yong Liu. On the minimum delay peer-to-peer video streaming: how realtime can it be? In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 127–136, Augsburg, Germany, September 2007. ACM.

[LL73] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[VJVS03] S. Voulgaris, M. Jelasity, and M. Van Steen. A robust and scalable peer-to-peer gossiping protocol. In *The Second International Workshop on Agents and Peer-to-Peer Computing (AP2PC)*. Springer, 2003.

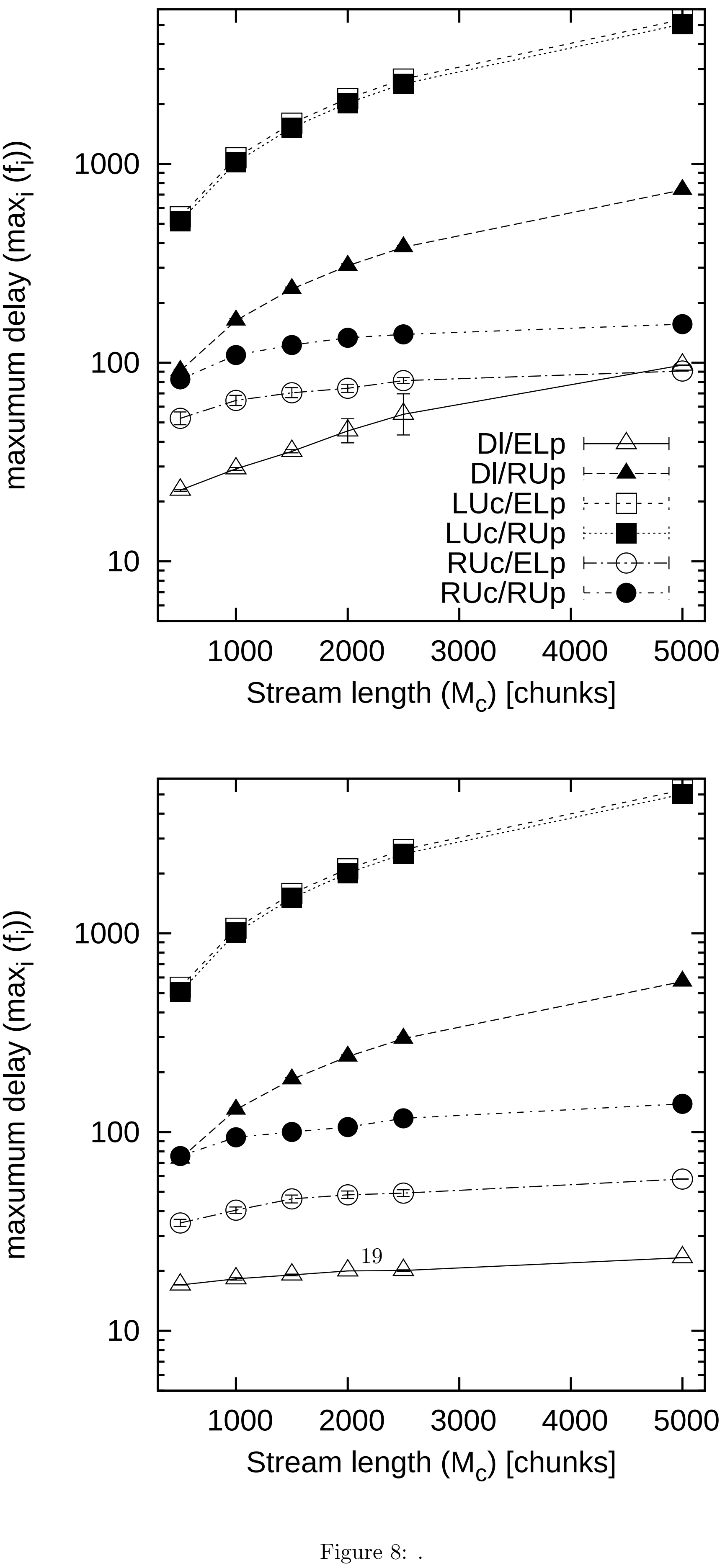
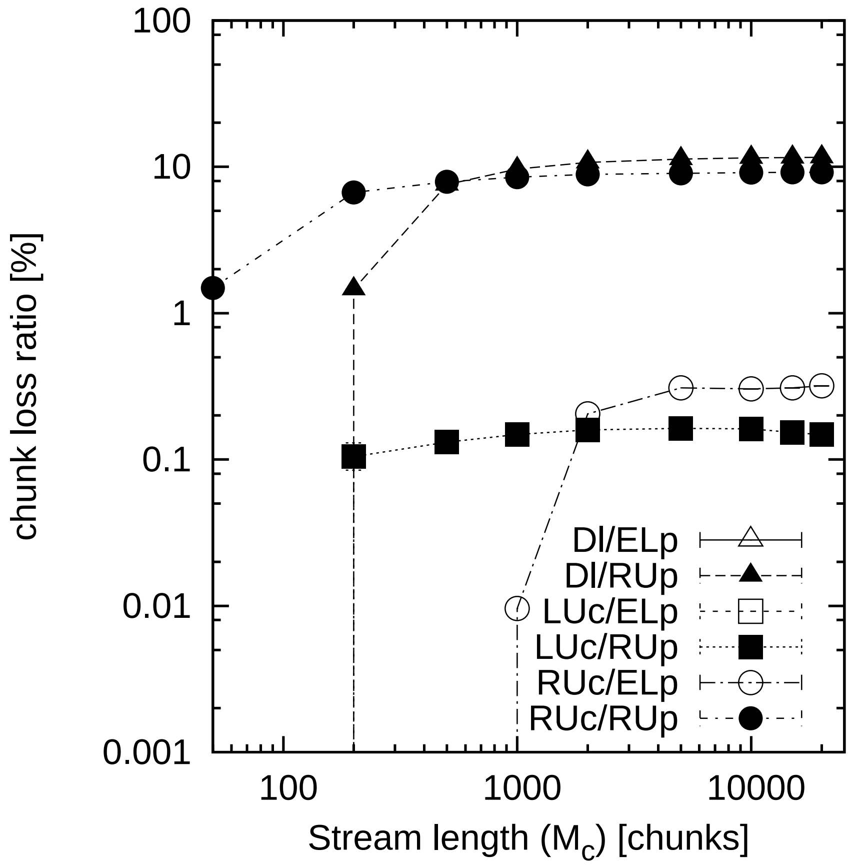

maxumum delay ($\max_i (f_i)$)
1000
100
10
1000
2000
3000
4000
5000
Stream length (M_c) [chunks]
Dl/ELp
Dl/RUp
LUc/ELp
LUc/RUp
RUc/ELp
RUc/RUp
maxumum delay ($\max_i (f_i)$)
1000
100
10
1000
2000
3000
4000
5000
Stream length (M_c) [chunks]

Figure 8: .



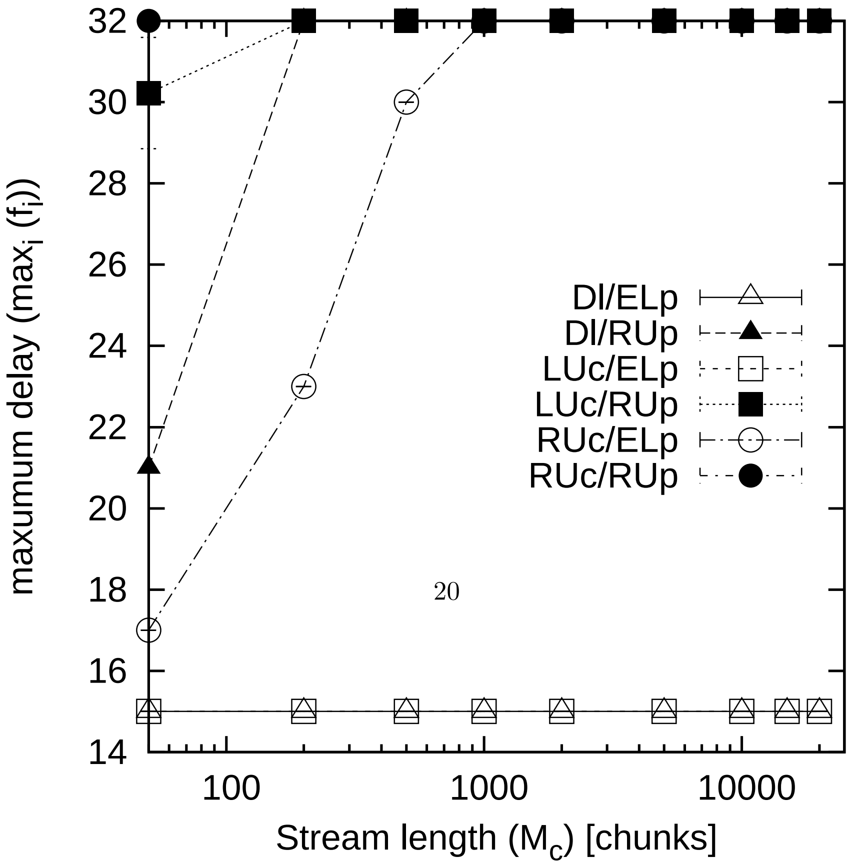


Figure 9: Full mesh overlay; chunk loss and worst case diffusion time as a funcion

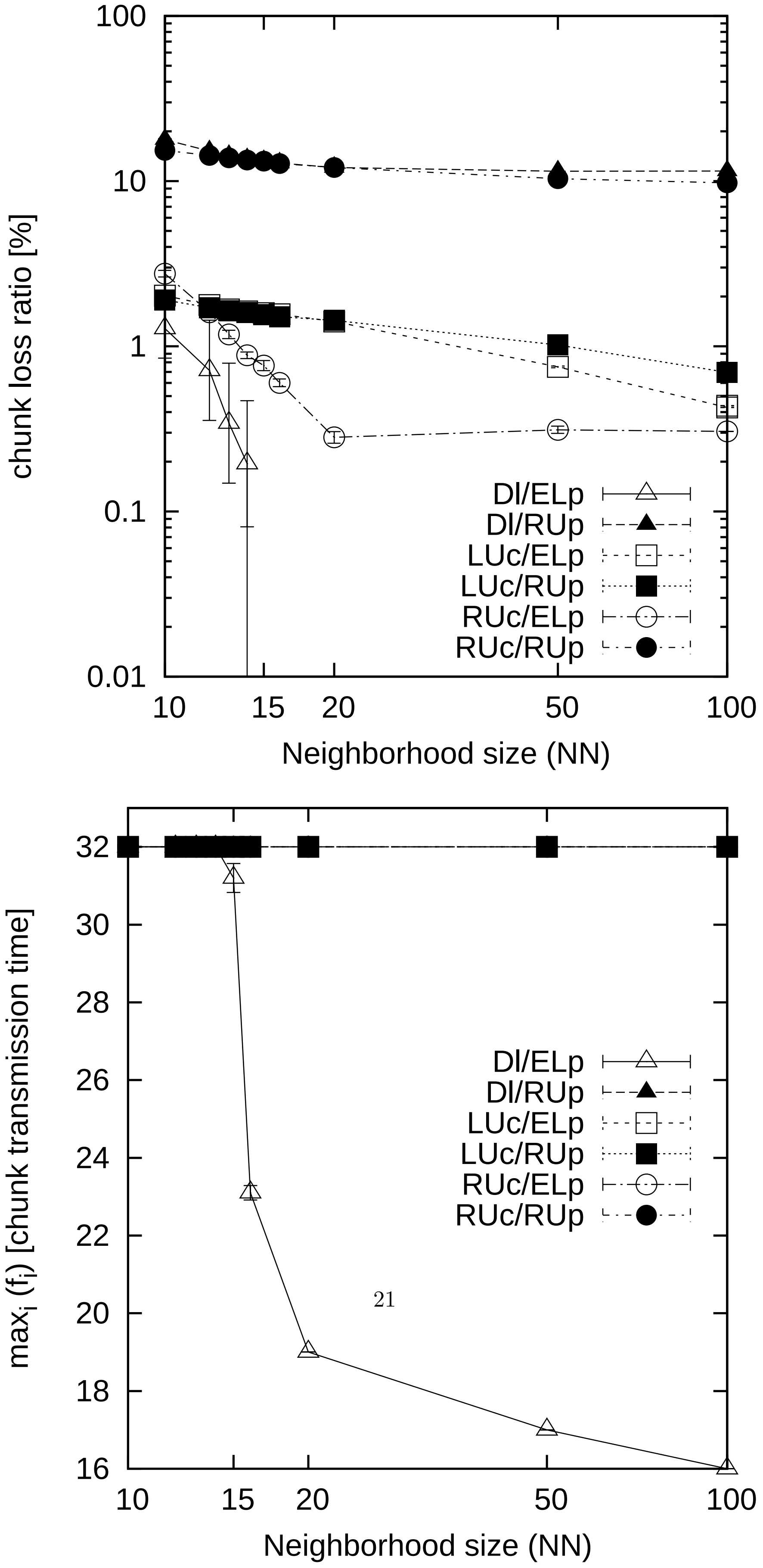


Figure 10: Chunk loss ration as a funcion of the neighbourhood size ($N =$ 10000, $B = 32$)