©2007 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The definitive version of this paper has been published in Workshop on Modeling, Simulation and Optimization of Peer-to-peer environments (MSOP2P) in conjunction with Euromicro (PDP 2007), 2007, 10.1109\/pdp.2007.41.

Efficient Simulation of Large-Scale P2P Networks: Compact Data Structures

Andreas Binzenhöfer*, Tobias Hoßfeld
University of Würzburg
Institute of Computer Science
Germany

Gerald Kunzmann
Technical University of Munich
Institute of Communication Networks
Germany

Kolja Eger
Hamburg University of Technology (TUHH)
Department of Communication Networks
Germany

Abstract

One of the most important design goals of current Peer-to-Peer (p2p) technology is to be able to offer its service to an arbitrary large number of users. Discrete event simulation is often applied to quantitatively and qualitatively evaluate the performance and scalability of such systems before they are deployed. However, the number of users, processes and events which can be simulated is limited by both the central memory and the time available. In this paper we present compact data structures and event design algorithms, which are intended to be a further step towards efficient simulation of large scale p2p systems. In particular, we give guidelines on how to increase the number of peers which can be simulated and show how to find a good trade-off between computational time and memory consumption in large scale p2p simulation.

1 Introduction

The algorithms and methods of Peer-to-Peer (p2p) technology are often applied to networks and services with a high demand for scalability. In contrast to the traditional client/server architecture, 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 such p2p services and their corresponding networks, different possibilities like emulation, analytical approaches, or simulative techniques can be applied.

Planetlab [5], e.g., offers an overlay network testbed which currently consist of more than 670 worldwide distributed machines. While it can be used to experiment with prototypes or to perform a proof-of-concept, it does in general not suffice to evaluate new algorithms at a larger scale. The huge number of peers, the state space, as well as the interactions and relationships between peers and states also makes an analytical description intractable. Discrete event simulation on the other hand is able to incorporate all interactions and parameters and may reflect reality as accurately as possible. Such simulations, however, also require sufficient memory capacities and exceed the computational power very fast. It might already be a problem just to keep the states of the peers in the main memory.

Simulations on packet level, e.g. using ns-2 [13], usually are too detailed for the evaluation of large overlay networks and do not capture effects which only happen at a larger scale. Therefore special p2p simulators like PeerSim [9] have been developed with extreme scalability and support for dynamic user behavior in mind. Those simulators can be used to easily implement and evaluate new algorithms but do not fully take advantage of the special features of the individual algorithms. A good overview of current tools to simulate p2p networks can be found in [3].

In general there are several possibilities to improve the implementation of a p2p simulation. To speed up the simulation, the computation process can be distributed to a cluster of PCs. Such a distributed simulation engine, which is able to run millions of instances, is, e.g., presented in [7]. A two step approach to simulate p2p networks on different layers is introduced in [2]. In the first step the network is simulated on application level, while a more detailed packet layer analysis can be done in the second step in order to consider such parameters as loss and delay. In this paper we share our own experience and findings gained during

*Corresponding author: binzenhoefer@informatik.uni-wuerzburg.de

the simulation of large scale p2p networks. In particular, we discuss the influence of the event queue, show how to reduce the amount of necessary memory and point out how to exploit the special features of events like periodic updates of routing tables in p2p networks. The main goal of these efforts is to be able to simulate very large overlay networks in order to capture effects, which are only visible at a larger scale.

The remainder of this paper is organized as follows. Section 2 defines the term simulation efficiency. In Section 3 we discuss how to apply efficient data structures like a calendar queue to large scale p2p simulations. Section 4 deals with the design of periodic and dynamic events. The concept of process handlers to improve the state representation in p2p simulations is described in Section 5. Section 6 concludes the paper and points to associated work.

2 Categories of Simulation Efficiency

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;
- Design of events 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. 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

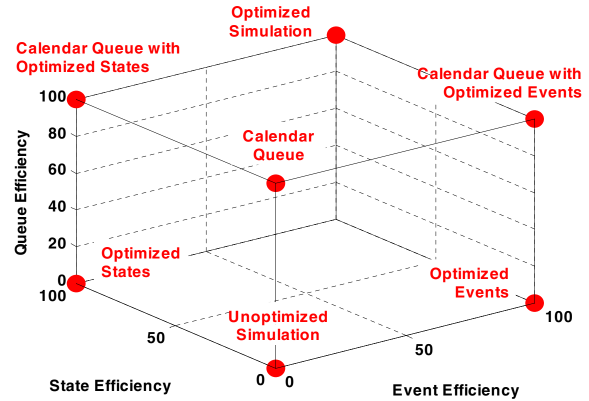

Figure 1. Different categories of simulation efficiency.

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 3 discusses the advantages and disadvantages of a special priority queue when applied to the P2P environment. We will present possibilities to adapt the queue to the specific features of a P2P system. Since the performance of the queue depends on the number of events and their temporal distribution, we point out the importance of event design algorithms in Section 4. Using Kademlia bucket refreshes, we will show how to model periodic and dynamic events efficiently. In Section 5 we will introduce a novel approach to reduce the memory required 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.

3 Compact Data Structures

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 [11], 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 structured P2P networks are mainly used as information mediators, a simulation usu-

ally 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. Structures like trees and heaps have an improved hold time of $O(log(n))$.

An optimal solution 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 [4], a queue with a hold time of $O(1)$.

3.1 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 [12]. There are numerous proposals to realize efficient priority queues [10]. In this section we show how a basic calendar queue [4] 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 2 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 is not limited to 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

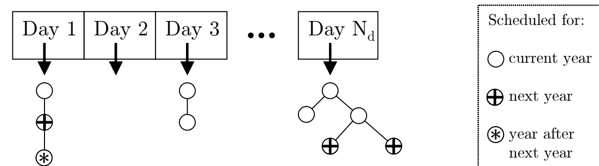

Figure 2. A simple calendar queue.

$$index = \left\lfloor \frac{timestamp}{T_d} \right\rfloor + 1 \pmod{N_d},$$

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.

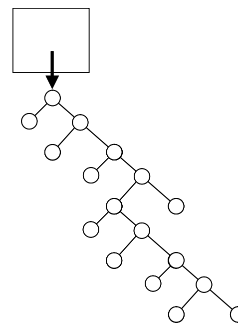

Figure 3. A day with too many events increases the enqueue time.

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 3. 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 4), the dequeue time will raise, as a lot of days without any event have to be examined until the next event is finally found.

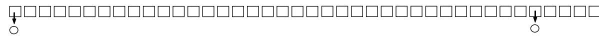

Figure 4. Too many days increase the dequeue time.

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 [1]. An efficient way to restructure the calendar queue on the fly can be found in [12].

3.2 Calendar Queue in P2P Simulations

To study P2P specific effects on the calendar queue, we simulate a Kademlia [8] 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 such a way that the average number of peers stays 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. Figure 5 represents a snapshot of the calendar queue, showing all 4096 days on the x-axis and the corresponding number of events scheduled at each day on the y-axis.

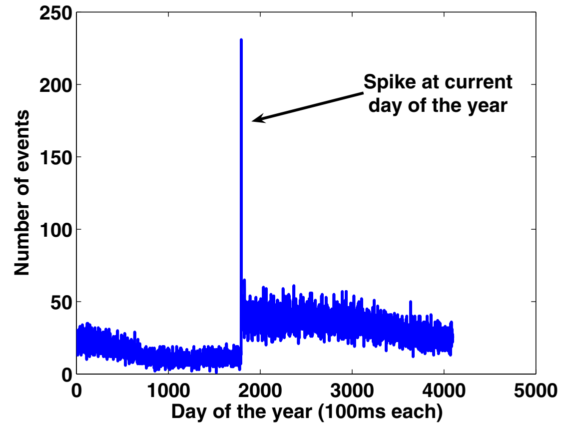

Figure 5. Snapshot of a calendar queue with $N_d = 4096$ and $T_d = 100$ms.

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 (short time scale).
- Periodic events, like the stabilize mechanism in Chord.
- Events that take place in a more distant future, like timeouts or bucket refreshes in Kademlia (large time scale).

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 50ms 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 5 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 6.

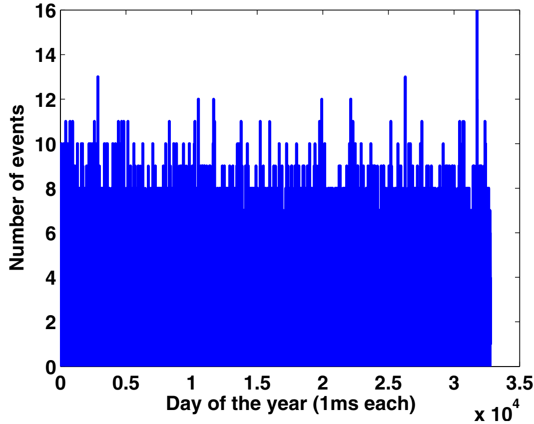

Figure 6. Snapshot of a calendar queue with $N_d = 32768$ and $T_d = 1$ms.

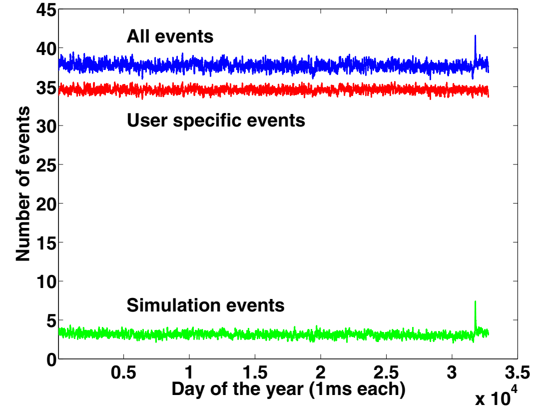

Figure 7. Composition of the events in the calendar queue.

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 approximately $O(1)$.

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.

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 7 illustrates this problem in detail. For the sake of clarity we plotted the moving average with a window size of 40.

The top most curve shows the distribution of all events in the event queue. The events can be split into those read from the input file (user specific events) and those generated during the simulation (simulation events). 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 7 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 $O(1)$.

4 Event Design Algorithms

The previously discussed performance of the event queue is 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,

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.

4.1 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 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. Therefore, the following section discusses how to handle dynamic events efficiently.

4.2 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 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 number of peers in the overlay network. Each of these 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.

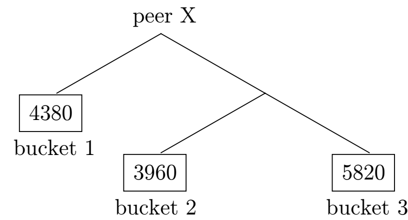

Figure 8. The next refresh times of three exemplary Kademlia buckets.

Figure 8 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 9. 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, drastically increases the execution time, since deleting an event from a calendar queue requires $O(n)$ steps.

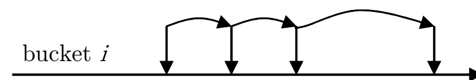

Figure 9. Refresh event moved every time the peer uses the bucket.

To reduce the running time, we should therefore invest some computer memory. For each 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 events from the queue, however, they are simply skipped when being dequeued as indicated by the dotted arrows in Figure 10. 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.

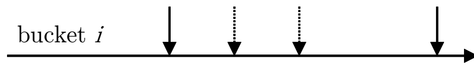

Figure 10. Obsolete refresh events are being skipped.

This solution, however, requires more computer memory than actually necessary. Especially if there are a lot of searches 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 11. This way, the memory needed to store the obsolete refresh events can be avoided entirely. 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!

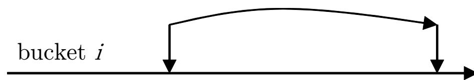

Figure 11. Obsolete refresh events are completely avoided.

Considering that bucket refreshes can only be moved forward in time, we can develop an optimized solution in terms of required 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. 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 12.

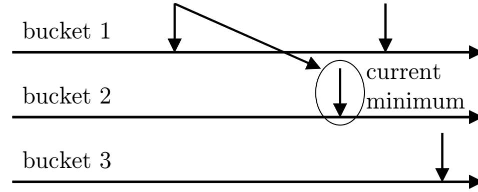

Figure 12. Refresh event scheduled at minimum next refresh times of all buckets.

Since this procedure takes exactly one hold time, it 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 8. 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 is scheduled at the current minimum 4380.

5 State Representation

To achieve scalability for large scale p2p simulations, the cost to represent a peer or a resource must be as low as possible 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.

In large scale p2p simulations computer memory is almost as equal a 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 description 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.

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 13 shows a process handler with $R_a = 3$ remaining accesses, as there are still three events

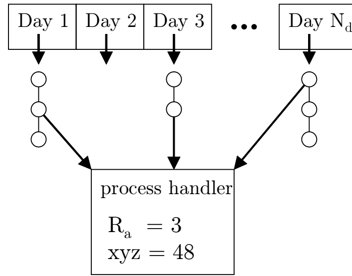

Figure 13. Example of a process handler with 3 remaining accesses.

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.

6 Conclusion and Associated Work

Many properties of highly distributed p2p systems only become observable when the number of participating peers is sufficiently large. In this paper we investigated different possibilities to make the simulation of large scale p2p systems feasible. We showed how to adapt the parameters of a particular event queue to the special features of a p2p environment. We also discussed how to approach the trade off between running time and memory consumption. Using a sophisticated design for periodic and dynamic events, both of which are common to p2p systems, we were able to reduce the amount of required memory while still maintaining the same functionality. Finally, we exploited the redundancy of distributed processes which include more than one peer. The concept of a process handler avoids the use of unnecessary memory by managing the shared information of parallel processes.

The simulation of p2p overlay networks is a very complex task which includes numerous other aspects which we could not regard here. This paper only represents a part of our work dealing with the efficient simulation of p2p networks. The corresponding technical report [6] addresses many other questions like whether to simulate on packet or on application level, how to model bandwidth in filesharing systems or how to simulate realistic delays for $O(n^2)$ end-to-end connections.

Acknowledgments

The authors would like to thank Prof. Phuoc Tran-Gia, Prof. Jörg Eberspächer, and Prof. Ulrich Killat for enabling and supporting this work. Furthermore, we would like to thank Robert Henjes and Holger Schnabel for the numerous remarks and ideas, as well as their help and support in implementing the various algorithms.

References

[1] J. Ahn and S. Oh. Dynamic calendar queue. In *SS '99: Proceedings of the Thirty-Second Annual Simulation Symposium*, page 20, Washington, DC, USA, 1999. IEEE Computer Society.

[2] H. Birck, O. Heckmann, and R. Steinmetz. The Two-Step P2P Simulation Approach. *Journal of Communication Software and Systems (JCOMSS)*, 1(1):4–12, Sept. 2005.

[3] A. Brown and M. Kolberg. Tools for peer-to-peer network simulation. Internet-Draft Version 00, IETF, January 2006.

[4] R. Brown. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM*, 31(10):1220–1227, 1988.

[5] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):00–00, July 2003.

[6] T. Hoßfeld, A. Binzenhöfer, D. Schlosser, K. Eger, J. Oberender, I. Dedinski, and G. Kunzmann. Towards efficient simulation of large scale p2p networks. Technical Report 371, University of Würzburg, October 2005.

[7] S. Lin, A. Pan, R. Guo, and Z. Zhang. Simulating large-scale p2p systems with the wids toolkit. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 415–424, Washington, DC, USA, 2005. IEEE Computer Society.

[8] 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.

[9] PeerSim: A Peer-to-Peer Simulator. http://peersim.sourceforge.net/.

[10] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul.*, 7(2):157–209, 1997.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, August 2001.

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

[13] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.