Simulating Large-Scale P2P Systems with the WiDS Toolkit

Shiding Lin[†] Aimin Pan[†] Rui Guo[‡] Zheng Zhang[†]
slin@microsoft.com *aiminp@microsoft.com* *guorui@sei.buaa.edu.cn* *zzhang@microsoft.com*
Microsoft Research Asia[†] *Beijing University of Aeronautics and Astronautics*[‡]

Abstract

Current simulation technologies support at most hundreds of thousands of nodes, and fall short on the emerging large-scale networking systems that usually involve millions of nodes. We meet this challenge with our distributed simulation engine that is able to run millions of instances and is tested with a production P2P protocol, using commodity PC clusters. This simulation engine is part of the WiDS toolkit, which takes a holistic approach to the research and development of distributed systems. We also propose a critical optimization, called Slow Message Relaxation (SMR), to trade simulation accuracy for performance. By taking advantage of the fact that distributed protocols are resilient to network fluctuation, SMR executes events in a logical time window much wider than the conventional lookahead scheme allows. We analyze and bound the potential effect of the distortion on application logic and other general metrics. Our experiments demonstrate that the simulation engine is able to achieve order of a magnitude speedup with statistically accurate simulation results.

1. Introduction

Simulation is the key to building and understanding of distributed systems. It plays a crucial role at different stages of the development process. For instance, coarse-grained simulation helps to understand the system at a crude level, allowing us to verify the assumptions and study various metrics before the detailed protocol is derived. We can also run and test the protocol logic against logic simulation, and crystallize the protocol incrementally. When the protocol is finally implemented, it is important to make the same code base run-able in a simulator and deployable in the real network. This is so because the development process is often iterative, as we find problems and seek optimizations and then test. Simulation helps to debug and evaluate the protocol in a controllable environment.

The P2P network such as Chord [1], Pastry [2] and Tapestry [3] typically involve millions of dynamic nodes, and pose a great challenge for the research community to come up with a systematic methodology. To this end, we have built the WiDS toolkit that tackles the difficulties at different development stages and unites the whole process in a holistic solution [4].

The general philosophy of WiDS can be summarized as "*code once and run many ways*". It provides several runtimes to run a protocol in different modes, targeting at different phases of the development process. Its simple, message-passing based APIs isolate the protocol from any particular runtime and make the environment switch transparent. In simulation mode, the events are stored and dispatched in the event queue(s), and a customizable topology model can be employed to enforce the latency. The simulation runs on a single machine, or on a cluster of machines. Single machine simulation allows debugging a distributed protocol within one address space, whereas parallel simulation enables the understanding of the protocol in a large scale. In the network execution mode, WiDS provides a socket-based library, yielding a system ready to run in the networking environment. WiDS users maintain one single code base throughout the development process, and invoke different runtimes by simply re-linking against different libraries. We are also working on a replay facility that logs the events in the network execution mode and replays them in the simulation mode, thus enabling postmortem debugging of a distributed system. We have built many protocols and systems using WiDS, including one very large scale storage system that uses commodity storage bricks [5].

One critical component of WiDS is its large-scale simulation engine. We have also done extensive testing with a scale of millions of instances for the Peer Name Resolution Protocol (PNRP) [6] based on the production code from Microsoft, using hundreds of clustered PCs. To our knowledge, this is one of the largest simulations that have ever been attempted.

In this paper, we will present our simulation engine used in WiDS. Our novel contributions are the followings:

♦ We develop a new protocol for distributed event simulation that utilizes commodity PC cluster. It adopts the master-slave architecture with a practical and simple protocol, and is efficiently implemented. With hundreds of machines, we are able to simulate millions of protocol instances.

♦ We propose and implement the slow-message relaxation (SMR) optimization. It executes the events in a logical time window much wider than the lookahead allowed by minimum network delay, taking advantage of the fact that distributed protocols are designed to be resilient to network fluctuation. Thus, it reduces the synchronization overhead and improves the simulation performance significantly. Although SMR is designed for a family of P2P overlays, the idea can be generalized to study other applicable large-scale distributed systems.

- ♦ This optimization, however, needs to be carefully applied, or otherwise simulation accuracy can be severely compromised. We analyze and bound the slow message effect on application logics and other general metrics. The experimental results verify our analysis.
- ♦ Guided by the bound analysis, our adaptation scheme dynamically adjusts the relaxation window according to runtime information. This derives optimal performance with minimum perturbation on accuracy, as is demonstrated through our experiments.

The remainder of the paper is organized as follows. Section 2 discusses related work. We describe our basic protocol and architecture in Section 3. Section 4 details the SMR optimization and its protocol. Section 5 analyzes its effect. We present the experimental results in Section 6 and conclude in Section 7.

2. Related work

Simulation of large-scale system is notoriously difficult [7], yet systems such as P2P overlays have to be studied close to their anticipated scale, otherwise facing die consequences when deployed. In fact, our experience of simulating the PNRP protocol is such that there are problems only discoverable at a scale close to millions. This is the dilemma that we are facing.

Parallel and distributed simulations [8] have been developed for many years and there are several simulation packages, such as pdns [9], SSF [10], USSF [11], GloMoSim [12] and GTNetS [13]. Their core engines can be divided into two categories, conservative and optimistic. In the conservative approach, all LPs (Logical Process) advance the logical time in a coordinated manner and the simulation is carried out round by round; in each round LPs synchronize with each other and process only the safe events in the window allowed by minimum network delay commonly referred to as the *lookahead*. Thus, the chronicle order of event execution is guaranteed. However, the main problem of the conservative approach is its excessive synchronization overhead, especially when the scale is large. The optimistic approach, known also as time warping [14], is entirely different in that each LP advances its logical time independently. When an *old* event with a timestamp less than current logical time is received, which means the chronicle order is violated, the LP rollbacks to the event's time by restoring the state and recalling out-sent events. It is clear that the optimistic approach needs to save a lot of states and the cost of cascading rollback can be very high. This is especially problematic when studying large-scale system. Our simulation engine is based on the conservative approach, but improves it in a number of significant ways.

The first improvement is related to how synchronization is done. The full-barrier synchronization described in [15] has O(N^2) cost by letting each LP flood to all the others. The Critical Channel Traversal (CCT) [16] takes advantage of the property that only those LPs with safe events need to run. The safe events can be found by traversing all the input channels and finding out those critical ones, whose source LP has advanced its time to the timestamp of the first event in the channel. By iteratively resolving the critical channel, the simulation proceeds round by round. However, this assumes that some global state is accessible by each LP, and in a cluster environment the scheme falls back to O(N^2) flooding. Although broadcasting capability is available in the LAN environment and can be leveraged as is done in [17], dedicating one machine as master and employing the simple master-slave architecture seems more versatile and achieves the same level of message cost and availability. Moreover, our protocol handles the transient messages in the destination side, instead of waiting for them in all peers [17], and thus performs a little better.

The second is on how to improve performance with minimum sacrifice on accuracy. Our work is aimed at simulating a completely developed protocol stack, as such some of proposals (e.g. selective abstraction [18]) are not directly applicable. Other schemes focus on complementary aspects in the network layer [19][20]. Instead, we observe that, any properly design distributed protocols have timeout logics that tolerate network fluctuation. We proactively take advantage of this fact and extend the logical time window, which can often be several magnitudes larger than the lookahead. Upon receiving an old event, we simply replace its timestamp with current logical time and process it, as if the event suffers from some additional network delay. By carefully bound the relaxation window, we show that statistically accurate results can be obtained. Furthermore, the window size can be dynamically adjusted to achieve optimal speedup. The optimization of relaxing the exact timestamps is also considered in [21][22]. The difference is that we leverage the domain knowledge of distributed systems and protocols, carefully analyze the impact of relaxation to application logic and derive the bound within which no rollback is necessary. Although our evaluation is performed over P2P applications, our technique is generally applicable to a broad class of distributed systems. Another optimization considered by [23] is to schedule the events by the order in which they arrive into the queue, not the chronicle order of their timestamps. Obviously, this optimization is not appropriate for networking protocols because the timer events and remote messages might be treated incorrectly.

Federation is a methodology to coordinate multiple instances of sequential simulators. It is reported in [13] that million-node scale can be achieved in a super-computer with thousands of processors. Though the tightly-coupled architecture contributes a lot to the performance, the achievement is significant. Our system can also be regarded as a federated architecture with homogeneous simulation engines, and the performance and scale we have achieved will justify the applicability of realistic large-scale simulation on commodity PC clusters.

Emulation [24][25] is another popular mechanism to study large-scal systems. Protocol instances run unmodified on a cluster of machines, and each one may host many instances. The latency is either statically specified or dynamically simulated using a routing core. In the later case the problem becomes, once again, on how to simulate a large-scale system (i.e. the routing core). At any rate, imposing the exact latency on the communicating protocols is difficult: the load on the sending and receiving machines and the fluctuation occurring in the physical components such as switches are all hard to be accounted for beforehand. One possible remedy is to slowdown the physical clocks of the machines. It will be interesting to see how to adjust the clock dynamically. Yet, what is clear is that old messages exist and they need to be handled one way or the other.

The critical difference of these approaches lie in how time is managed, especially with respect to the way delay model is enforced. Simulation strictly enforces the model in logical time, and the optimistic approach indirectly fulfills the same goal with rollback. Emulation adopts physical time but may relax by ignoring the uncontrollable delay due to physical situation. SMR, on the other hand, enforces the logical time and relaxes with a bound.

3. The basic protocol and architecture

We are aiming at large-scale distributed protocol simulation which may involve millions of protocol instances. It is impossible for one single machine to afford such demanding computation and memory resources, so we have to turn to distributed solutions. The commodity PC cluster seems to be a good fit.

Our design is guided by simplicity, low overhead and high performance. We will use the term "node" to denote one simulated instance. On each physical machine, instead of embodying each node in a *run-able* thread, we adopt an event driven architecture. Events of all nodes, usually messages and timers, are aligned in an event queue in the timestamp order. There is one LP associated with all the nodes on a *slave* machine. LP_i's local clock, noted as LVT_i, is always equal to the timestamp of the head event in its event queue. A *master* coordinates all the *slave* LPs. This leads to a two-level architecture shown in Figure 1.

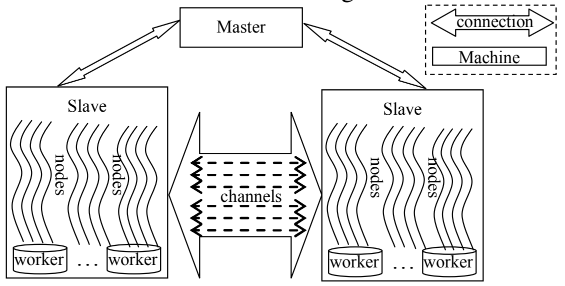


Figure 1. The two-level architecture. Each slave LP as well as the master is hosted by a physical machine. Channels for each pair of nodes are multiplexed in the pre-established connections between machines. To harness the hardware parallelism such as SMP or hyper-thread, we employ multiple worker threads, each of which takes charge of a sub-group of nodes.

Right after its generation in LP_i, a timestamped event *e* is delivered to its destination LP_j and merged to its local event queue. Its timestamp TS_e is calculated by $TS_e = LVT_i + d_e$, where d_e is the latency of the event, specified by a network model. We let be the globally minimum value of *d*, i.e. the global lookahead.

The protocol is rather intuitive, and the core idea is to guarantee the chronicle order. Each LP can only process those *safe* events, which are defined to be those whose timestamps fall in [*GVT*, *GVT*+), where *GVT* is the globally least clock among LPs. The *critical* LPs are the subset of the total LPs that have safe events for a given *GVT*.

At the very beginning, every LP reports to the mater its *LVT*, and the master computes the *GVT* and the critical LPs. The master then informs those critical LPs of the *GVT* contained in an **EXEC** message. Accordingly, the critical LPs start to run till *GVT*+ . The execution of this round has not only changed the *LVT*s of the critical LPs themselves, but also generated events that could change *LVT*s of other LPs as well. Thus, after finishing a round of execution, a critical LP sends the master a **SYNC** message, which includes its new *LVT* and a list recording the timestamps of the events it has sent to any other LPs. This allows the master to compute both the *GVT* and the critical LPs for the next round. However, the reception of **EXEC** messages from the master alone is only a necessary but not sufficient condition for safe event execution. This is because an event from LP_i to LP_j may arrive later than the **EXEC** message from the master, which is common in the network environment where triangle inequality no longer holds. Therefore, the master has to act as a gate-keeper to track the number of events for LPs, and tell LP_i in the **EXEC** message the number of safe events that LP_i must wait before executing the events. The count of safe events for LP_i can be calculated by $C_i = \sum_{j\in N} M_{j,i}$, where $M_{j,i}$ is the number of events sent from LP_j to LP_i with timestamp in [*GVT*, *GVT*+) – this is why the **SYNC** message from LP_j needs to contain the timestamps of the messages it sent to other LPs.

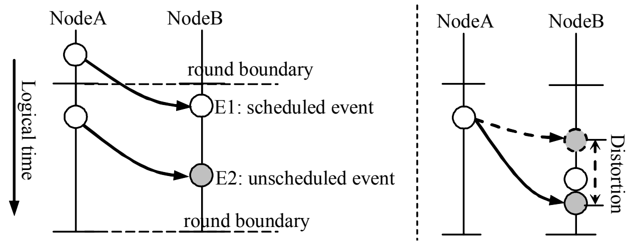
Our partial barrier is efficient in that only the critical LPs need to be involved in the synchronization, which is separated from simulated data transmission. Messages are directly transmitted to their destinations and are processed as soon as they are allowed to, guaranteed by the control info maintained by the master. To some extent, our approach can be treated as an aggregation of the flooding in the full-barrier-based synchronous approach described in [15], reducing the number of messages from $O(N^2)$ to $O(N)$.

We want to clarify the availability issue regarding centralized architecture such as master-slave. It is an illusion that the availability can be improved without a centralized controlling master. In all the past proposals, the crash of any one of the LPs will halt the simulation. Thus, master-slave architecture does no worse. In fact, we have augmented our protocol to be fault resilient. If a master crashes, we can bring up a new master and reconstruct its state from the slaves. If a slave crashes, the master will eliminate it from the slaves and allows the rest to move on. The later case is acceptable to P2P overlay simulation because this is as if a group of nodes have left the system. The detailed analysis of this issue is available in a separate technical report [26].

4. The Slow Message Relaxation

The barrier model becomes increasingly inefficient when the number of machines in the cluster increases. Our approach to improving performance is to reduce the number of barriers in a simulation run. The optimization is called Slow Message Relaxation (SMR) that basically extends the simulation window from [*GVT*, *GVT+*) to [*GVT*, *GVT+R*), where *R* is the relaxation window. Thus, for each barrier, we will execute more than safe events in a round.

This brings about two issues. First, although we can still track the *scheduled* events, which are generated in the previous rounds, there will be some other events that are generated on the fly. Some of these on-the-fly events will become scheduled events in the future rounds, and others will have timestamps in [*GVT+* , *GVT+R*) and must be processed in the current round. Such events are called *unscheduled* events. We need a mechanism to guarantee that all unscheduled events – the total number of which is unknown a priori – are processed in this round. The technique we developed is called the *quantum barrier*, and will be discussed in greater detail in Section 4.2.



(A) The scheduled, unscheduled events. (B) A slow message.

Figure 2. During the simulation, many on-the-fly events will be generated. Those with timestamp in the current round are called unscheduled events, while those across round are scheduled events. If an unscheduled event falls behind the current clock of the destination upon its arrival, it is turned into a slow message and its latency is changed.

The second issue is more subtle. Scheduled events can be guaranteed to be executed in chronicle order and at their timestamps. But there is no such guarantee for an unscheduled event. It is possible that an LP receives an unscheduled event whose timestamp is behind its clock. Such a message is called a *slow message*. The conventional way is to avoid the handling of such messages by rolling back. We argue that this is not necessary. The reason is that if we replace the slow message's timestamp with the current clock, then from the simulated protocol point of view it is as if the message had suffered from some extra delay in the network. A properly designed distributed protocol should have already handled any network-jitter generated abnormality. It is for this reason that we call this optimization the Slow Message Relaxation. As we will show later (Section 5), by taking advantage of the fact that a distributed protocol must be able to tolerate network uncertainty, the relaxation window can be significantly wider than what the conventional lookahead window can allow (often at the range of hundreds). On the other hand, in such a window there is noticeable percentage of slow messages, and the use of roll-back will result in practically unacceptable performance.

Of course, if the time relaxation is used too aggressively, the simulation results can be severely distorted. Thus, the relaxation window should be carefully selected, and ideally should be adaptive to the simulation run. We show this analysis in Section 5. The concepts of scheduled, unscheduled events and the slow message are depicted in .

4.1. The SMR Protocol

The pseudo code for SMR protocol evolves from the basic protocol, and is written in the asynchronous message handling fashion as shown in Table 3 and Table 4.

Like the basic protocol, the LP is scheduled to run by the **`EXEC`** message from the master, which contains *GVT*, GVT_{UB} and C_i. *GVT* is the minimum value of *LVT*s and GVT_{UB} represents *GVT+R*. C_i is the number of scheduled events of LP_i, and is calculated by $C_i = \sum_{j \in N} M_{j,i}$, where $M_{j,i}$ is the number of events sent from LP_j to LP_i with timestamp in [*GVT*, GVT_{UB}). If all the scheduled events are received, the LP can start executing the events till GVT_{UB}. When an unscheduled event arrives, whose timestamp is less than GVT_{UB} (line 16), it is processed immediately. When all the events in the execution window are processed, the LVT_i and M_i are sent back to the master in the individual **`SYNC`** message, upon which the master is able to calculate a new *GVT* and C_i. When the master notices that all the unscheduled events have been received and processed, it proceeds to the next round.

```
1.  Run:
2.      while Queue.head.ts < $GVT_{UB}$ do
3.          get the head event and remove it from Queue
4.          $LVT_i$ := max($LVT_i$, Queue.head.ts); // ensure that time never goes back
5.          process the head event, for each event it generates
6.              deliver the event to its destination $LP_j$ and update $M_{i,j}$
7.      send SYNC message to the master, with $M_i$ attached
8.  end.

9.  OnExecMsg(GVT, $GVT_{UB}$, $C_i$):        // the EXEC message from manager
10.     $LVT_i$ := GVT;                        // update logical time
11.     if all (Ci) sched. evts have been received and Queue.head.ts < $GVT_{UB}$ then
12.         Run;                    // execute those events that have arrived
13. end.

14. OnReceiveExternalEvent(event):
15.     Queue.insert(event);
16.     if all ($C_i$) scheduled events have been received and event.ts < $GVT_{UB}$ then
17.         Run;
18. end.
```

Table 3. The pseudo code for the slave LP.

```
19. OnSyncMsg($M_i$):          // SYNC message from $LP_i$
20.     merge $M_i$ into M
21.     if all the events in [GVT, $GVT_{UB}$) have been received and processed then
22.         calculate the new GVT, C and R, according to the M
23.         send EXEC message to all LPs
24. end.
```

Table 4. The pseudo code for the master.

Next, we will present the algorithm that guarantees all the unscheduled events will be processed, and then discuss how to automatically derive the appropriate R for each round.

4.2. Quantum barrier

The difficulty of the SMR protocol is to guarantee all the unscheduled events, which are generated on the fly, be received and processed within the current round, i.e., to guarantee the completeness of events in the barrier window [GVT, $GVT+R$). We call such barrier the *Quantum Barrier*.

If we treat each event as a node and the derivation relationship between events as a link, we will get a run-time spanning tree of events for a round, with *leaves* representing the events that do not generate any unscheduled events. If we define the execution of an event as the *access* to the node, our problem can be abstracted to be a distributed traversal algorithm of a spanning tree that is generated at runtime. Due to the lack of a global state, the tree traversal problem seems not as easy as that in the centralized environment.

A naïve solution will work as follows. For a tree, we know that the sum of the fan-out degree of all nodes is the number of tree nodes plus 1. Thus, when we access a node, we can report its fan-out degree, i.e. the number of its children, to a central repository. Similarly, we can also report to the repository the number of processed events. Thus, the barrier is reached when these two numbers are equal. This is obviously not optimal.

The tree traversal terminates when all the leaf nodes are accessed. The problem is that we do not know how many leaf nodes exist in such a highly dynamic tree. We solve this with a simple trick using *tokens*. The central repository gives the root of a tree a token with a value, say 1. Iteratively, whenever a non-leaf event generates child events, it passes a split token to each child and the value is the current token's value divided by the number of children. All leaf events report their tokens back to the repository, and if the sum of these tokens equals to 1, it knows that the spanning tree traversal – and therefore the execution of all the events – is complete.

This simple division has two practical problems. First, the fan-out of an event can not be got a priori. In order to count the sum, the descendent events have to be buffered before being delivered to their destinations. This is not very efficient. Second, the token has its inherent limitation in precision; we need a more scalable representation.

Our solution to both problems is shown graphically in Figure 5. Essentially, the parent's token is split by half each time a new event is generated. Each token is represented by an integer i, representing its value of $1/2^i$, and thus avoids underflow.

Mapping back to our protocol, it is natural to employ the master as the repository to collect the tokens. Each critical LP is assigned a token with value 1 to start with. The master sums up all the tokens reported back from the slaves that have executed any events in the round, and if the sum equal to the number of critical LPs, the current round terminates. Since an LP will process multiple events, it is not necessary for every leaf event to report to the master, and instead the reports are aggregated and attached in the **`SYNC`** messages.

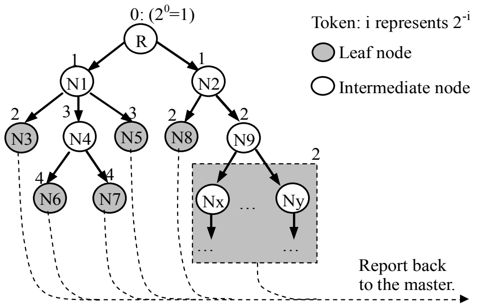


Figure 5. Token is employed in the distributed traversal algorithm for the run-time spanning tree. When the master collects enough tokens, it knows that all leaves have been accessed.

4.3. Runtime Adaptation

In the next section, we will discuss what the proper SMR bound of the window width should be so that we can attain statistically accurate performance result. It would seem natural to set R as large as possible for optimal performance gain. However, this turns out not to be true. Through numerous experiments we have found that the performance does not improve monotonously as R increases. One of the reasons is that network congestion causes packet drop and thus TCP retransmission. Therefore, the adaptation of R must take cue from run time measurement.

```
1.  CalculateNewR():
2.      If ($R_{curr} > T_{min}/2$)                    // just in case $T_{min}$ has changed
3.              $R_{next} := T_{min}/2$
4.              return $R_{next}$
5.      $s_{curr} := R_{curr} / t_{curr}$               // calculate the speed of current
6.      $s_{prev} := R_{prev} / t_{prev}$               // and previous rounds
7.      $r_s := (s_{curr} - s_{prev})/(s_{curr} + s_{prev})$   // compute rate coefficient
8.      If ($R_{curr} > R_{prev}$) $D := 1$            // compute the directional coefficient
9.      If ($R_{curr} < R_{prev}$) $D := -1$
10.     If ($R_{curr} = R_{prev}$) $D := 0$
11.     $R_{next} := R_{curr} + T_{step} * r_s * D + \tau$   // compute new $R$ for the next round
12.     If ($R_{next} > T_{min}/2$)                    // check against $T_{min}$ again
13.             $R_{next} := T_{min}/2$
14.      return $R_{next}$
15. end.
```

Table 6. The algorithm to calculate R for the next round.

The adaptation is run at the master as shown in Table 6. It is a straightforward hill-climbing algorithm which is carried out before a new round starts (line 22 in Table 4). The call to the function **`CalculateNewR`**() defines the R_{next} to be used next, which is broadcast to all slaves in the **`EXEC`** messages. In the adaptation algorithm, R_{curr} is the R value for the current round, and T_{min} is a bound imposed by the application and is collected from all slaves. As we will show in Section 5.2, $T_{min}/2$ is the bound that R should not exceed. Line 2-4 checks if T_{min} has changed and sets R_{next} right away within the bound if R_{curr} exceeds it. Line 5-6 computes s_{curr} and s_{prev}, which are simulation speed in current and previous round, respectively. The rate coefficient r_s in line 7 is a signed value in rang (-1, 1), and its absolute value reflects the rate of speed change in the recent rounds, relative to the raw simulation speed. The intuition is that the adjustment is made slower when we approach the optimal value of R. The direction coefficient D in lines 8-10 is important because the improvement of speed (i.e. $s_{curr} > s_{prev}$) could have been brought by either positive or negative adjustment of R, and we want to continue the trend if that is the case, or reverse it otherwise. Line 11 computes R_{next}, and here T_{step} is a constant, and τ is a random disturbing factor in the range of [$-T_{step}/3$, $T_{step}/3$] to avoid local minimum. It also serves the purpose of making the adaptation process active especially in the initial stage.

So far, we have described the algorithm in terms of adaptation purely against performance goal. Users can define other adaptation metrics such as percentage of slow messages and upper bound of extra delays. Our adaptation can adjust accordingly as well.

5. Analysis

The core idea of SMR is to increase the parallelism by reducing the amount of barrier operations. The net effect of SMR is that some random messages will be subject to some random extra delays. By themselves, any properly designed distributed protocols should have already handled any network-jitter generated abnormality. However, if there are too many slow messages and, more importantly, if application logics are significantly altered, then the simulation results will be severely distorted. Thus, it is important to understand the effect of SMR.

The context of this study is to speedup the simulation of very large-scale P2P networks. We will give a brief background of these systems first, and then analyze the bound of SMR that we can afford. The introduction of SMR also changes the distribution of network latency. Since the probability of having a slow message is small with moderate window, we have found that the distortion of latency distribution is not significant. For completeness, the theoretical analysis is provided in our technical report [26].

As we will demonstrate in the evaluation section that follows, setting a correct bound ensures us to gain statistically correct results with large speedups. Whether SMR can be applied to other application logics is one of our future works.

5.1. Structured P2P networks

We are interested in simulating the so-called structured P2P such as Chord [1], Pastry [2] and Tapestry [3]. They are often called DHT (for *distributed hash table*) because a collection of widely distributed nodes across the entire Internet self-organize into a very large logical space (e.g. 160bit). These protocols represent a family of distributed systems that are very complex in nature. Because of their unprecedented scale (e.g. millions), there is an urgent need to understand their behavior at a large scale. Today, such system is often simulated using one machine, only reaching a tiny fraction (e.g. 5K nodes) of its anticipated scale.

Each node in DHTs has a random logical ID. Typically, there are several layers of application logics in each node. The bottommost layer essentially sorts these nodes into a linear array in a very large logical space. This layer uses the *leafset* routing table, which records a few closest logical neighbors (e.g. 2 nodes on each side). The leafset logic sends heartbeats to the leafset entries to ensure the continuity of the logical ring. On top of this layer, the *finger* protocol selects O(log$N$) nodes into a finger table, each of the fingers is exponentially further away. Here, N is the total number of nodes in the system. The fingers allows lookup – the upper layer application operation that requests the owner of a key in the space – to zoom in onto the destination in O(log$N$) steps, whereas the leafset ensures lookup termination.

Although implementations may differ, the protocols critically depend on the correct execution of timer logics. Some timers are static, and others are dynamic. The finger maintenance protocol is static: periodically (e.g. every 5 minutes) a node will refresh its fingers by firing probing messages to the current fingers. Stale entries are replaced by new ones which are found by issuing lookups of appropriate logical points in the space and recording the new owners. The refresh of leafset follows a similar fashion but can be implemented using dynamic timer instead. Every time a node hears from one of its neighbors, it starts a timer. If the timer expires before the next heartbeat arrives, the leafset logic enters into a waiting state with yet another timer. If the node misses several heartbeats in a row, it will suspect that the neighbor is dead and then take appropriate actions. Such protocol is a typical implementation of failure detection, which is fundamental to membership protocols. Notice that the correct interleaving of arrival order of heartbeats from *different* nodes is not critical, because they will settle at different entries of the routing table (for both finger and leafset). If the routing tables are maintained correctly, then statistically speaking upper layer applications such as user initiated lookups or node churns should not be impacted.

5.2. The effect on application logics

Let $T_{timeout}$ be the timer interval. The firing and timeout of a timer are two distinct events. It is clear that these two events can not be in the same simulation round; otherwise they may be simulated back-to-back without waiting for the action associated with the firing to have its effect. Thus, we must have $R < T_{timeout}$. To give an idea on how much relaxation this brought, note that $T_{timeout}$ can be in the order of seconds (even minutes), whereas a lookahead defined by a network model is often in the range of tens of milliseconds. With the typical configuration, this means the affordable window can be *several hundreds times* of a lookahead.

The problem of a slow message, in terms of the timer logic, is that it can generate false time-out. To understand this, we must first analyze the delay bound of slow messages. Refer to (B), if at t_0 an event generates a message whose delay is d, then the message will have a timestamp of t_0+d. If t_o+d is greater than the ending time of the current simulation round, the message becomes a scheduled event in some future round and there is *no* extra delay. Thus, the maximum extra delay happens when t_0 equals the beginning of the round, and upon arrival at the target node where the clock is one tick shorter than R and the extra delay will therefore be $R–d$. As a result, we can draw a conclusion:

Bound 1: The upper bound of extra delay of unscheduled events is $R–d$, where d is the minimum network delay.

Consider a two-step message sequence, for example, node A sends a message to node B, and B sends a second message back to A as a response. If both messages are slow messages, then they must be within the same round, therefore the extra delay will not exceed $R–2d$. If one of these two messages is slow message, then the extra delay will not exceed $R–d$. If both are not slow message, then no extra delay occurs. As a result, we can determine another upper bound of extra delay of slow messages:

Bound 2: The upper bound of extra delay of a two-message sequence is $R–d$.

Now let's look at how to choose R to avoid false timeout. Assume that A sends a request to B and starts a timer with the interval $T_{timeout}$, and then enters a waiting state. The round trip between A and B is $T_{round} = 2d$. If $R \le d$, there will be no distortion. So we only need to discuss the case when $R > d$. First, as a reasonable setting, $T_{timeout}$ should be larger than T_{round} in order to keep the timeout logic working with normal network delay. Based on the Bound 2 (the case of two-step sequence A->B->A), if $T_{timeou}>T_{round} + (R–d)$, then distortion does not lead to a false timeout. However, if $T_{round} < T_{timeou} \le T_{round} + (R–d)$, a false timeout might happen.

From $T_{timeou} > T_{round} + (R–d)$ and $T_{round} = 2d$, we can have

$$R < T_{timeout}–d$$

Since $T_{timeout}> T_{round}= 2d$, or equivalently, $d < T_{timeout}/2$, it follows that $R < T_{timeout}/2$ is a sufficient condition. As a result, if we set R to satisfy

$$R < T_{timeout}/2 \qquad (1)$$

Then distortion will not break the timer logic for request-response protocols. Other timer logics can be similarly dealt with. Detailed treatment is given in the technical report [26].

What we have discussed are the DHT logics. A DHT application will issue lookups, which may take O(log$N$) steps. How do we make sure that there is no false lookup timeout? In fact, we can extend the 2-step message bound to a k-step message sequence. When k>3, we can decompose a k-step message sequence as $k/2$ two-step message sequences, where the last combination may have only one message. The application programmer typically estimates a reasonable one-step network latency, adds some leeway, and times a conservative value of total hops (e.g. 2log$N$ for a reasonable N), and finally arrives at a lookup time-out setting. To be consistent, the two-step, request-response timeout value $T_{timeout}$ should also be used as a base to set the lookup timeout. Thus, $R < T_{timeout}/2$ will prevent false lookup timeout as well.

As we can see from the above analysis, though the DHT protocol is very complex, the bound $R < T_{timeout}/2$ is enough to keep the application logic as close as an undistorted simulation would achieve.

6. Evaluation

Our system's performance, scale and accuracy are functions of the number of nodes, the number of slaves and R. In our experiments, we chose a typical P2P DHT, XRing [27], to evaluate our results. We have also simulated the PNRP protocol [5] up to 1.5 million of nodes over more than 250 PCs. At such a large scale, we notice that the master starts to become performance bottleneck.

6.1. Experiment setup

XRing [27] is a typical structured P2P DHT, with a hybrid protocol of Chord [1] and Pastry [2]. The leafset protocol records a few logical neighbors, using a variant of the Pastry leafset protocol. Leafset members send heartbeats among each other. Fingers are maintained in a Chord style, with periodical refreshing.

In order to calculate the arrival timestamps for each message, WiDS slaves should be aware of the underlying network model so that the proper delay of each message can be imposed. In our experiments, the delay follows a uniform distribution between [1ms, 200ms]. In XRing, the leafset heartbeat cycle is 5s and a stale entry is removed when 3 heartbeat intervals have passed. The finger refreshing cycle is 15s, and a stale finger is discarded after 3 refresh cycles. Application lookup timeout is set to 30s. These values are typical settings of a wide-area P2P network. We take the minimum heartbeat timeout value (5s) and derive the bound of R to be 2.5s. As can be seen, this is a much larger relaxation window than the lookahead would allow. In fact, the uniform distribution means that the minimum lookahead is 1ms.

We have a 33-machine cluster in our lab, the CPUs are Pentium IV 3.0GHz, and each machine is equipped with 512M RAM. We configure one of the machines as the master and up to 32 slaves. All machines are connected by two 100Mb Ethernet switches. The underlying OS is Windows XP. Due to the limitation of physical memory, each machine can host at most 10,000 nodes; otherwise the frequent page-swapping will lead to dramatically performance degradation. In following experiments, the number of nodes on each slave will vary from 512, 1024, 2048, 4096, to 8192. On the other hand, we change Rs to verify the effect of SMR on simulation performance and distortion.

We ran the XRing protocol for 10-minute simulated time, divided into three phases. All nodes joined the XRing within the first 2 minutes, and the system settles in the next 3 minutes. The system should converge to a ring during the 2nd phase at some point. At the beginning of the 6th minute, each slave starts sending one lookup request to a randomly picked node every second. In this phase, i.e. the period from 6th to 10th simulated time, at every second each slave chose one node randomly and crashed it. The slave then generated a new node to join XRing. We designed such a simulation scenario because it is important to look at the system performance under node churns. This experiment setting is also typical in studying the dynamic behavior of P2P systems.

We present the results based on three sets of experiments. First, we ran a large-scale simulation with various *R*s and looked into the performance and the application logic metrics with respect to *R*s. Second, we ran two sets of simulations with fixed total numbers of nodes and fixed numbers of nodes on each slave to investigate the scalability of WiDS. Finally, we ran XRing with *R* being adapted at runtime.

6.2. Performance and application logics

First of all, we ran a large-scale simulation with 32 slaves, and each slave hosts 2048 or 8192 nodes. When *R* varies, the real execution time is shown in Figure 7.

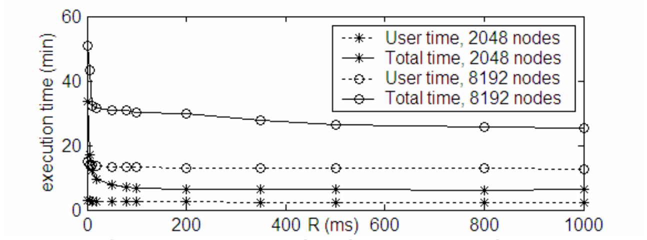


Figure 7. The execution time when varying *R*.

In the above figure, the dashed curves are the user time, i.e. the event handling time in slave machines. The solid curves are the total execution time, which includes user time, message exchanging time and event schedule overhead. Obviously, the total execution time decreases quickly as *R* increases. The user time is stable in either case, as is expected. But when *R* is beyond a few hundred milliseconds, there is essentially no gain. The speedups of the 2048 and 8192 nodes are 2 and 5.4, respectively. One reason of different speedups is that, when there are more nodes, there is more parallelism within a slave machine already, so larger *R* does not improve parallelism observably. Another reason is that with the same networking capacity, less nodes means less networking traffic for both synchronization and message exchanging, so more traffic can be aggregated, and then SMR with the larger *R* will be more effective.

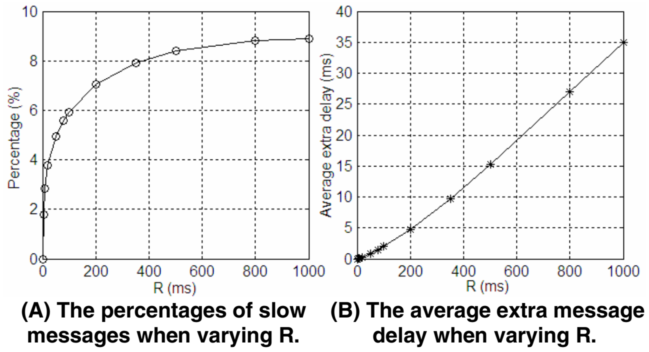
In addition to the execution time, we also observed the application logics when varying *R*. Since XRing is a DHT-based membership protocol, we measured its convergence time, which should appear somewhere during the period from 2nd to 5th simulated time. Another two metrics of application logics are the success rate of lookups and the average number of hops for the successful lookups. The results of the experiments with each slave hosting 8192 nodes are shown in Table 8, where *R* is chosen as 1, 50, 200, 1000, 16000, and adapted. According to XRing logics, if *R* does not exceed leafset timeout interval (15s), the leafset logic will not be broken; otherwise false-timeout will probably occur due to slow messages. As shown in Table 8, when *R* is set to 16000ms, the convergence time in the settlement period is delayed to 288019, which shows that the leafset logic is observably affected by slow messages. At the same time, we notice that the metrics of the success rate and the average number of hops are also affected when $R = 16000$. Because the lookup timeout is set to 30s and according to XRing protocol the average number of hops is $(\log_2 N)/2 + 1 = 10$, where N is 262,144, some of the lookups will be false timeout if there are some hops along the path suffering from an extra delay (the maximum possible extra delay is $R\text{-}d\approx 16$s), thus the success rate decreases. In this situation, the lookup with small hops to the destination node will survive with higher probability, therefore, the average number of hops decreases as well.

R	1	50	200	1000	16000	Adaptive
The time of convergence	228001	228001	228001	228001	288019	228001
Success rate of lookups	0.93	0.9297	0.9307	0.9297	0.5043	0.9301
Average hops	10.18	10.19	10.20	10.19	3.80	10.19

Table 8. Application logics when varying *R*.

Here our focus is not so much on offering an explanation what statistics are affected when the relaxation is overly aggressive, but on the fact that, indeed, when the relaxation is appropriately applied, statistically correct results can be obtained. From Table 8, we can see that when $R < 2500$ms, the application metrics are not affected by slow messages. This means that we can let WiDS choose *R* automatically using adaptation algorithm described in Section 4.3. The last column in Table 8 displays the corresponding metrics when *R* is adaptive.

Figure 9 shows the average extra delay and percentage of slow messages when varying *R*. Obviously, when *R* grows, the percentage of slow messages increases, and the extra delay increases linearly. Here the percentage of slow messages is relevant to the probability of a message being overly delayed (see the analysis in [26] for more information). We can see that when $R = 200$, the percentage of slow messages is about 7%; and when $R = 50$, it is less than 5%. This rate, while small, implies that there can be significant amount of rollbacks if an optimistic approach is used instead.

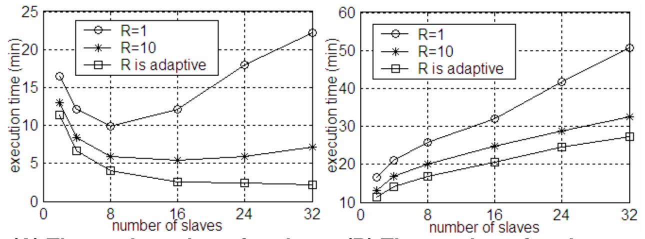


(A) The percentages of slow messages when varying R. **(B) The average extra message delay when varying R.**

Figure 9. The distortion metrics.

6.3. Experiments on scalability

Next, we examine the scalability of WiDS through two sets of experiments. We let *R*=1, 10ms or adapted by the algorithm described in Section 4.3. In the first set of experiments, we test how the performance would scale if more resources are available. We do this by fixing the total number of nodes while varying the number of slaves. The result is shown in Figure 10(A). The total number of nodes is 2*8192, or 4*4096, or 8*2048, or 16*1024, or 24*683, or 32*512.



(A) The total number of nodes is fixed. **(B) The number of nodes on each slave is fixed.**

Figure 10. The performance when the number of total nodes is fixed.

Generally speaking, when the number of slaves increases, the user load on each slave will decrease because the number of nodes on each slave decreases. However, the communication overhead between slaves and the master will increase. Therefore, given the total workload, when the number of slaves reaches a certain amount, these two effects may cancel each other out. Beyond a sweet-spot, adding more slaves gives diminishing return or can be counterproductive. Furthermore, since SMR optimizes the synchronization overhead, the sweet-spot is a function of *R*. For instance, the optimal number of slave is 8 and 16, for *R*=1 and *R*=10, respectively. On the other hand, adaptive *R* is robust as it always gives the best performance for a given cluster size.

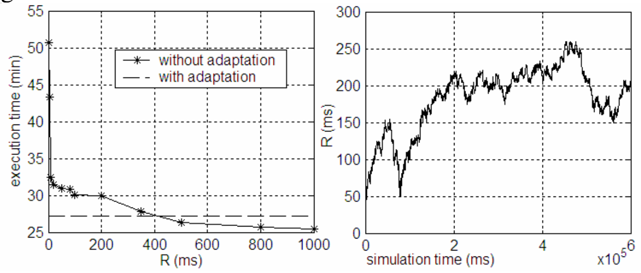
Another variable for optimal performance is application workload. Generally speaking, if we set *R*=1ms, the number of slaves is chosen to make each slave host as many nodes as possible without incurring excessive memory swapping. This is not entirely trivial, since other states in the simulation engine such as event queue(s) also consumes a certain amount of memory. A rule of thumb is to let the nodes take half of physical memory.

In the second set of experiments, we investigate the scaling of problem size. We do this by varying the number of slaves, while keeping the number of nodes on each slave fixed as 8192. In the most ideal case, the curves of execution time should be flat. This is not true for several reasons. First, the workload of application logic increases (e.g. the lookup path increase logarithmically with total number of nodes in the system). Second, communication overhead increases, too, with more slaves. This is especially true for configuration where communication overhead is less optimized (i.e. small *R*). Again, adaptive *R* is robust in the sense that its curve rises slowly (Figure 10 (B)).

In summary, SMR with adaptive window is robust and scalable. It achieves good speedup for a fixed problem size with larger cluster size. We use the sum of the user time on the two slaves in the 2*8192 configuration to estimate the minimum simulation time on a single machine. It shows that with 32 slaves, the speedup is about 12. We also perform well when scaling the problem size, a 16-fold increase only causes a 2.4 times slowdown.

6.4. Experiments on adaptation algorithm

Intuitively, when the duration of a round (i.e. *R*) increases, the performance will get better if the effect of slow messages are not concerned. But in practice, we find that in some workloads such as PNRP, increasing *R* will lead to performance degradation, as is described in Section 4.3. In fact, for any specific workload pattern there is a relatively optimal *R* value, i.e. the point where the application workload on each node matches the amount of networking traffic. If *R* is beyond this point far away, the performance will degrade instead.



(A) The performance when R is adaptive. **(B) The distribution of R during a run.**

Figure 11. The experiment results of the adaptation algorithm.

The adaptation algorithm described in Section 4.3 can be used to adjust *R* parameter automatically at run time to make it close to the optimal value under the current workload pattern. Figure 11(A) shows the result of the adaptation algorithm in the case of 32 slaves and each slave hosting 8192 nodes, where the dashed line represents the total execution time when *R* is adaptive. We can see that when *R* is adaptive, the overall performance is very close to the optimal. Figure 11(B) shows the changes of *R* during the simulation run.

7. Conclusion and future work

We present a new conservative protocol for parallel distributed event simulation. It is based on the practical master-slave architecture and can scale to millions of nodes with a moderate size PC cluster. We have implemented a slow-message relaxation optimization, which executes the events in a much wider logical time window than the conventional lookahead scheme allows. It takes advantage of the fact that most distributed protocols have already had leeway in timeout setting to handle network fluctuation. An adaptation scheme is proposed to dynamically adjust the width of the relaxation window according to runtime information so that optimal performance can be achieved with minimum perturbation on accuracy. Our experimental results show that the SMR optimization is effective and the distortion is negligible.

We are exploiting more optimization opportunities for our protocol. One of the possibilities is to adopt smart node placement scheme which partitions the simulated network topology such that frequently communicated nodes lie in one LP so as to reduce the message exchange cost. We are also trying to accommodate more instances of simulated protocols in one physical machine. The basic idea is to store protocol states and events in the disk, and swap them into memory when needed.

Acknowledgement

We would like to thank Noah Horton, Geogy Samuel, Brian Lieuallen and Sandeep Singhal for the support of running large-scale simulation of the PNRP protocols. We also thank the anonymous reviewers and Perry Zheng for their insightful inputs.

References

[1] I. Stoica, R. Morris, D. Karger, et al, “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”, SIGCOMM, 2001.

[2] A. Rowstron, P. Druschel, “Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems”, in IFIP/ACM Middleware, 2001.

[3] B.Y. Zhao, J. Kubiatowicz, A.D. Josep, “Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing”, UCB Technical Report No. UCB/CSD-01-1141.

[4] Shiding Lin, Aimin Pan, Zheng Zhang, et al, “WiDS: an Integrated Toolkit for Distributed System Development”, in HotOS X, 2005.

[5] Zheng Zhang, Qiao Lian, Shiding Lin, et al., “BitVault: a Highly Reliable Distributed Data Retention Platform”, under submission.

[6] Microsoft TechNet, “Introduction to Windows Peer-to-Peer Networking”, http://www.microsoft.com/technet/prodtechnol/winxppro/deploy/p2pintro.mspx

[7] G. Riley and M. Ammar, “Simulating large networks: How big is big enough?”, in Conference on Grand Challenges for Modeling and Simulation (ICGCMS), 2002.

[8] A. Ferscha, and S.K. Tripathi, “Parallel and distributed simulation of discrete event systems”. Technical report, University of Maryland, August 1994.

[9] G. Riley, R. M. Fujimoto, and M. A. Ammar, “A generic framework for parallelization of network simulations”, in MASCOTS, 1999.

[10] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski, “Towards realistic million-node internet simulations”, in International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 1999.

[11] D. M. Rao and P. A. Wilsey, “Simulation of ultra-large communication networks”, in MASCOTS, 1999.

[12] X. Zeng, R. L. Bagrodia, and M. Gerla, “GloMoSim: a library for parallel simulation of large-scale wireless networks”, in Workshop on Parallel and Distributed Simulation (PADS), 1998.

[13] Richard M. Fujimoto, Kalyan S. Perumalla, et al., “Large-Scale Network Simulation: How Big? How Fast?” in MASCOTS, 2003.

[14] David Jefferson, Brian Beckman, et al., “Distributed Simulation and the Time Warp Operating System”, in SOSP, 1987.

[15] D.M. Nicol and R. Fujimoto, “Parallel simulation today”, Annals of Operations Research, pages 249-285, 1994.

[16] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary, “Scheduling critical channels in conservative parallel discrete event simulation”, in Workshop on Parallel and Distributed Simulation (PADS), 1999.

[17] George F. Riley, Richard Fujimoto, Mostafa H. Ammar, “Network aware time management and event distribution”, in Workshop on Parallel and Distributed Simulation (PADS), 2000.

[18] P. Huang, D. Estrin, and J. Heideman, “Enabling large-scale simulations: selective abstraction approach to the study of multicast protocols”, in MASCOTS, 1998.

[19] Benyuan Liu, Daniel R. Figueiredo, Yang Guo, et al, “A Study of Networks Simulation Efficiency: Fluid Simulation vs. Packet-level Simulation”, in IEEE INFOCOM, 2001

[20] Syam Gadde, Jeff Chase, Amin Vahdat, “Coarse-Grained Network Simulation for Wide-Area Distributed Systems”, in Communication Networks and Distributed Systems Modeling and Simulation (CNDS), 2002.

[21] Martini P., M. Rumekasten, and J. Tolle, “Tolerant synchronization for distributed simulations of interconnected computer networks”, in Workshop on Parallel and Distributed Simulation (PADS), 1997.

[22] Loper M. and Fujimoto R., “A Case Study in Exploiting Temporal Uncertainty in Parallel Simulations”, in the International Conference On Parallel Processing (ICPP-04), 2004.

[23] Dhananjai Madhava Rao, Narayanan V. Thondugulam, et al., “Unsynchronized Parallel Discrete Event Simulation”, in the Winter Simulation Conference, 1998.

[24] Amin Vahdat, Ken Yocum, Kevin Walsh, et al, “Scalability and Accuracy in a Large-Scale Network Emulator”, in OSDI, 2002

[25] Emulab project. “The Utah network testbed” (Web site). http://www.emulab.net/.

[26] Shiding Lin, Aimin Pan, Rui Guo, Zheng Zhang, “Simulating Large-Scale P2P Systems with the WiDS Toolkit”, Technical Report, MSR-TR-2005-95, Microsoft Research, 2005.

[27] Zheng Zhang, Qiao Lian, Yu Chen, “XRing: Achieving High-Performance Routing Adaptively in Structured P2P”, Technical Report, MSR-TR-2004-93, Microsoft Research, 2004.