A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems

Shi-Ding Lin
Microsoft Research Asia
i-slin@microsoft.com

Qiao Lian
Tsinghua University
lq97@mails.tsinghua.edu.cn

Ming Chen
Tsinghua University
cm01@mails.tsinghua.edu.cn

Zheng Zhang
Microsoft Research Asia
zzhang@microsoft.com

Abstract— **Mutual exclusion is one of the well-studied fundamental primitives in distributed systems. However, the emerging P2P systems bring forward several challenges that can't be completely solved by previous approaches. In this paper, we propose the Sigma protocol that is implemented inside a dynamic P2P DHT and circumvents those issues. The basic idea is to adopt queuing and cooperation between clients and replicas so as to enforce quorum consensus scheme. We demonstrate that this protocol is scalable with system size, robust to contention, and resilient to network latency variance and fault-tolerant.**

I. Introduction

One of the fundamental primitives to implement more generic systems and applications on top of P2P DHTs [4][13] is mutual exclusion. Such primitive is also a rudimentary service needed by applications running on top to guard arbitrary resources when necessary. For example, a concurrency control mechanism is obviously needed for a mutable distributed file system.

For the applications and systems we envision to be built and deployed on those P2P DHTs, one can all but rule out the possibility of enforcing concurrency using stable transaction servers, whether they are external or internal to the system. Therefore, such primitives must be implemented *inside* P2P DHT. The protocol is thus by definition distributed, it must be simple and efficient, and yet robust enough to be of practical use.

Our basic idea is simple: utilizing the fact that nodes in the DHT collectively form a logical space that does not have holes, institute a set of *logical replicas* upon which a quorum consensus protocol grants access to critical section (CS). From a client's perspective, these replicas are always online. However, they may suffer from complete memory loss from time to time. Such random reset occurs when the node that acts as a logical replica crashes and gets replaced by one of its logical neighbor in DHT.

The open and dynamic nature of P2P environment brings another serious challenge. Many previous approaches [10] assume a close system with fixed and relatively moderate number of nodes, and nodes communicate among themselves to reach consensus. These solutions are inapplicable in our context where the number of clients is unpredictable and can swing to be very large. The protocols are designed for a harsh and open environment such as a wide-area P2P.

The work in this paper presents a few novel contributions:

- We start by investigating a straightforward, ALOHA-like strawman protocol and show that, the high variation of network latency between clients and replicas is responsible for the large performance degradation. We believe this insight is valuable for any wide-area consensus protocols.
- We demonstrate that a cooperative strategy between clients and replicas is necessary to circumvent latency variance and contention, thus achieving scalability and robustness.
- We propose the *informed backoff* mechanism, which intelligently rebuilds replica's state, to handle the random reset problem of replicas.

The resulting protocol, Sigma, is fully implemented, analyzed and evaluated. We present both analytical and experimental results that demonstrate its performance and efficacy.

We discuss the system model in Section II. The strawman protocol and its performance evaluation are presented in Section III. Building on this, we describe the Sigma protocol in Section IV and experiment results in Section V. Related work is in Section VI and we conclude in Section VII.

II. System Model

In its essence, a P2P DHT offers a virtual space populated by participating peers. The space does not have any holes except for a very transient period of time during membership change.

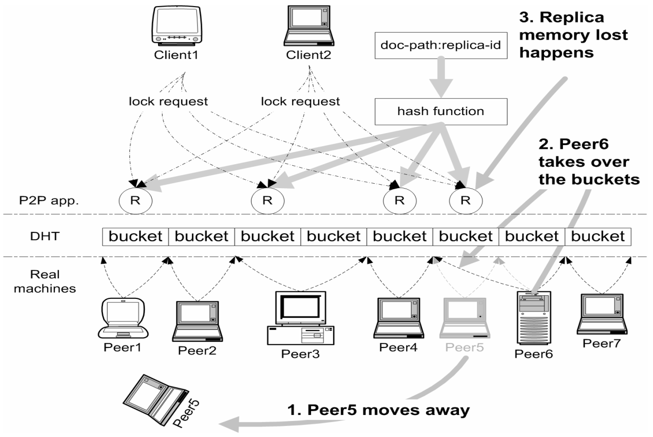

Figure 1. Majority consensus in a P2P DHT; "bucket" is the unit of DHT and is synonym of "zone", "R" denotes a logical replica. The diagram illustrates that the node crash can be modeled as logical replica reset.

For a given resource *R*, its associated server *CS*(*R*) can be *logical*. For instance, there can be *n* replicas whose names are "/foo/bar:*i*" where *i* є[1..n] (see Figure 1). We can hash these names to derive the keys, and the hosting node of each key can serve as one replica. This decoupling of naming and actual server means that we are working with a peculiar "server" who is *always* available but may suffer from memory loss at random point of time. Moreover, as we shall see, the introduction of multiple replicas implies that the latencies between a client to these replicas be highly variable, exerting a significant impact on performance.

Formally speaking, our system model is as follows:

- The replicas are always available, but their internal states may be randomly reset. This is also termed as the failure-recovery model in [8].
- The number of clients is unpredictable and can be very large. Clients are not malicious and fail stop.
- Clients and replicas communicate via message, but the channel between them is unreliable. Messages could be replicated, lost, but never forged.

In the context of this paper, both clients and replicas are peers in the DHT (in practice, however, it's only the replicas that have to be DHT members). When a replica grants permission (or vote) to a client, the latter is called *owner* of the former. A client who has collected majority permissions is said to be the *winner* of a round.

We assume that the typical lifetime of a DHT node is long enough so a client can talk directly to the current node who acts as a logical replica, invoking O(log*N*) DHT lookup only after the logical replica takes a reset.

Our primary goal is to derive a set of efficient and highly reliable protocols. We want the protocol to perform as robust as possible, in both low and high contention situation. Finally, it should correct itself rapidly after faults occur.

III. A Strawman Protocol

In this section, we will introduce a strawman protocol which is straightforward to implement, but nevertheless illustrates essential attributes as well as problems of a highly available, majority-based consensus protocol.

The main idea is similar to ALOHA [11] protocol's way of resolving conflicting packets; all clients that want to enter critical section (CS) send requests to each of the replicas and wait for responses. A replica grants a lease if it is not owned by anyone, and otherwise rejects the request but informs the client the current owner. The one who obtains *m* out of *n* replicas (*m>n/2*) is the winner at this round. Losers release acquired votes (if any), back-off and retry after a random period. This also guarantees that deadlock will not occur.

The replicas, however, can suffer from random reset after which it forgets about its previous decision and is open to new request. The "change of heart" can cause the mutual exclusion to be broken. Assume that average life of a node is *T*, the probability that the node may crash in a period of *t* is *t/T*. The probability that any *k* out of *m* voted replicas resets is $\binom{m}{k}(\frac{t}{T})^{k}(1-\frac{t}{T})^{m-k}$. So, the safety will be broken when more than or equal to 2*m-n* resets occur during t, and the probability would be $\sum_{k=2m-n}^{m}\binom{m}{k}(\frac{t}{T})^{k}(1-\frac{t}{T})^{m-k}$. It turns out that in a given round, to tolerate up to *k* replica reset *n=3k+1* and *m=2k+1* are needed. Thus, as a design choice, we can raise the value of *m*.

Figure 2 shows the probability of violating exclusivity, where *T* is of 10000 seconds, which is the reported average life of a P2P node [12], and *t* is chosen as 10 seconds, which gives a conservative upper bound of clients staying in the CS. We show the results with different parameters of *m/n*.

It can be observed that accomplishing robust mutual exclusion with this protocol is realistic. Even with *n*=32 and *m/n*=0.75 (i.e. *m*=24), a very practical setting, the chance of breaking the exclusivity is 10^{-40}. Using this configuration to guard a document whose availability requirement is 15-18 nines is entirely reasonable. The broader point we want to make is that it makes little practical sense to guarantee the conflict probability substantially lower than the availability of the resource to be protected.

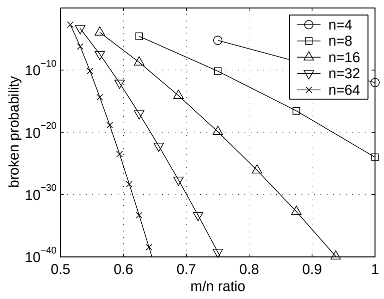

Figure 2. Probability to break exclusivity.

The performance of this protocol, however, is an entirely different story. Figure 3 depicts the throughput in various network conditions, where *m* and *n* is 24 and 32 respectively. The average network latency of any client to all replicas is fixed at 100ms, but latency of a given client-replica pair is a random variable. This is a reasonable assumption in a wide-area DHT.

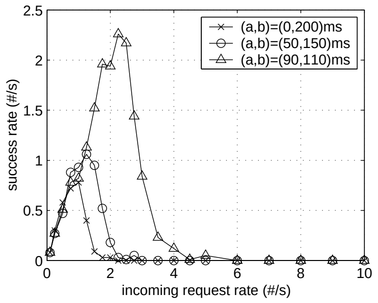

Figure 3. Performance of strawman protocol, with latency uniformly distributed in (*a, b*).

The first thing to notice is that the curve looks just like that of ALOHA(see [11]): the throughput increases linearly when contention is low, reaches a peak, and then degrades essentially to zero. It is clear that latency variance has a significant impact: the higher the variance, the worse the throughput.

IV. The Sigma Protocol

The main culprit of the strawman protocol's poor performance is the variance of network latency between one client and each replica. Client requests will reach different replicas at different time, so it is hard for all replicas to build a consistent view of competing clients. A second, more subtle issue has to do with is the greedy behavior of the clients: they would keep on retrying when collision occurs; therefore nobody can win out finally. There should be a comprehensive set of techniques to address both problems.

We deal with the first problem by installing a queue at replicas and reshuffle them towards a consistent view in case of high contention. To combat the second issue, we adopt a strategy to enforce clients into a state of *active waiting*.

Figure 5 and Figure 6 present pseudo code, described in terms of message handler, for clients and replicas respectively. The two sides exchange messages and the relevant entities and their interactions are depicted in Figure 4.

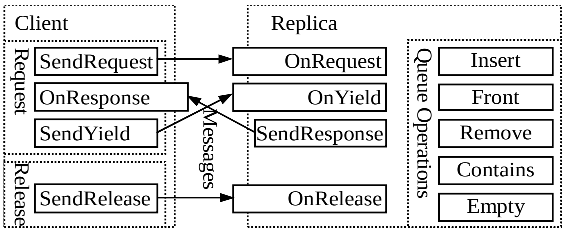

Figure 4. Architecture of Sigma protocol.

The client states include its `id`, and an array `resp[]`. `resp[i]` records the response from the *i*-th replica, stating its owner and the associated timestamp. The replica maintains the following: who owns this replica (C_{owner}), and a value of `nil` indicates this replica has not voted for anyone; T_{owner} stores the timestamp of the request; `Queue` stores waiting clients in order of their timestamp. We use Lamport's logical clock [7] to generate timestamp.

Client starts by firing REQUEST messages to all replicas (`Request` in Figure 5). These requests are handled by the `OnRequest` at the replica, which either grants its vote outright or queue the request up, depending on whether this replica has already voted. Regardless, the id and timestamp of the current owner of the replica (equals to the client's only if the queue is empty) are returned to client by a RESPONSE message.

As the responses arrive at the client (`OnResponse` in Figure 5), it gradually forms the idea about its place in the race. Suppose *m* out of *n* replicas is needed to achieve quorum consensus, there will be all but three outcomes and each can be easily distinguished by examining `owner` attached in the RESPONSE messages:

1. The client is the winner by quorum consensus. It succeeds and gets the permission to enter CS.
2. Some one does win but not this client. The client does nothing because it knows it has been registered on the replicas already. We will detail later when and how it will be notified.
3. *Nobody* has won, if $j_{same} + n - j < m$ where j is the number of returned responses and the maximal number of same item in j is j_{same}. The client then sends out a YIELD message to each of the acquired replicas.

```
State Variables:
    id      // the identity of the client
    resp[]  // responses from replica
Request(CS) {
  timestamp := GetLogicalClock();   //lamport's clock
  for each R[i] of CS
    SendRequest(R[i], id, timestamp);
}
OnResponse(R[i], owner, timestamp) {
  resp[i].owner := owner;
  resp[i].timestamp := timestamp;
  if (enough responses received) {
    winner := ComputeWinner();
    if (winner = self)        // case 1
      return success;
    else if (winner = nil) {  // case 3
      for each resp[i].owner is self {
        SendYield(R[i], id);
        Clear(resp[i]);       // reset the state
      }
    }
    // case 2: some one else wins, just wait
  }
}
Release(CS) {
  for all R[i] of CS
    SendRelease(R[i], id);
}
```

Figure 5. Client-side pseudo code.

```
State Variables:
    $C_{owner}$        // the client it accepts
    $T_{owner}$        // time stamp of Cowner
    Queue         // queue requests up
OnRequest(C, timestamp) {
  if (Cowner = nil) {
    $C_{owner}$ := C;
    $T_{owner}$ := timestamp;
  }
  else
    Queue.Insert(C, timestamp);
  SendResponse(C, Cowner, $T_{owner}$);
}
OnRelease(C) {
  if (C = $C_{owner}$) {
    $C_{owner}$ := nil;
    if (not Queue.Empty())
      RespQueue();
  }
  else if (Queue.Contains(C))
    Queue.Remove(C);
}
OnYield(C) {
  if (C = $C_{owner}$) {
    Queue.Insert(C, $T_{owner}$);
    RespQueue();
  }
}
RespQueue() {      // helper routine
  <$C_{owner}$, $T_{owner}$> := Queue.Front();
  SendResponse($C_{owner}$, $C_{owner}$, $T_{owner}$);
  Queue.Remove($C_{owner}$);
}
```

Figure 6. Replica-side pseudo code.

The YIELD operation reflects the collaborative nature of Sigma protocol and is a critical performance optimization. The semantic of YIELD is RELEASE+REQUEST. When replica receives a YIELD message, it removes the client from the winning seat and inserts it into the queue, chooses the earliest one and notifies the winner.

The function of the YIELD handling is to reshuffle the queue. The fact that nobody wins indicates that contention happens. This, in turn, implies that queues are being built up but the winners are out of place. By issuing the YIELD request, clients are collectively offering the replicas a chance to build a more consistent view and, consequently, choose the right winner. It is important to understand that this will go on until a winner is settled, as such could be multiple rounds of YIELD. Typically, this self-stabilization process will quickly settle.

The release operation is straightforward: the client, if is the owner simply relinquishes, or is removed from the queue otherwise. In either case, the next client (if any) is notified by the RESPONSE message.

So far, we have described Sigma in a failure-free environment. In reality, many things may go wrong:

1) After crash, a replica might grant the vote to a new client, despite the fact that it might have already done so to the previous one. We deal with this by raising *m/n* ratio to reduce the probability of breaking safety. With appropriate parameter, the risk can be negligible in practice (Section III).

2) More seriously, its queue has gone and those waiting clients will get stuck forever. What is called for is a way to rebuild the replica's memory. This is addressed by the *informed backoff* mechanism discussed later.

3) If the client who is currently in the CS crashes before exit, replicas will be stuck. Therefore, replica grants permission to clients with renewable lease [1]. When the lease expires, replica will grant permission to the next client (if any) in the queue.

4) The unreliable communication channel between a client and a replica will cause similar problems as well. In essence, message loss can be mapped to arbitrary crash of clients or replicas, which simplifies the handling.

The combination of informed backoff and lease builds a reliable communication over the unreliable channel and is a variation of failure detector [15] plus timeout. This best-effort approach leverages replica knowledge to achieve better tradeoff between communication cost and system throughput.

Informed backoff is a way to rebuild a restarted replica's state without overloading those healthy replicas. It is extremely simple at its core. Upon a request, replica could predict the expected waiting time T_w and advise it to wait so long before next retry. An empirical calculation of T_w is $T_w = T_{CS} * (P + 1/2)$, where P is the client's position in the queue and T_{CS} is the average CS duration, as observed by the replica of interval between any two consecutive release operations. The 1/2 in the formula is to take current owner of the replica into consideration. Notice that T_w is always updated upon the reception of a retry.

Let's consider the case that the replica has not crashed at all. If the client is notified before its scheduled retry, no harm is done. Otherwise it means that the advised T_w was not accurate (such as some earlier clients take extra time). In this case the client will renew its T_w in its retry. This is of course an overhead, but hopefully we are not too far off from the future point when the permission will be granted and thus hopefully this is the only retry that the client will have to endure. If, on the other hand, replica does go through a reset, then the queue is reconstructed with the order similar to the original one, fulfilling our goal.

We now offer a brief analysis of the Sigma protocol:

- **Service policy**. The use of logical clock and the First-Come First-Serve policy at replica does not guarantee FCFS, since client requests can take arbitrary long to arrive. Thus, Sigma can be best described as quasi-FCFS.
- **Safety**. We guarantee safety with high probability. No known protocols can ensure 100% correctness under failure. We treat replica failure and imprecise failure detector in a uniform manner. As shown in Section III, the probability of violating safety can be practically negligible by setting appropriate *m/n*.
- **Liveness**. Progress is ensured by using lease.

V. Experimental Results

The Sigma protocol is fully implemented and deployed in a distributed testbed, which can be configured by different network topology models.

We assume a pool of infinity clients, and each client will fire request contending for CS according to a Poisson distribution, λ of which is the incoming request rate. To focus on the performance aspects of the protocol, we let client exit CS immediately after it enters. After 5 minutes warm-up period we test 10 minutes during which throughput, in terms of the number of serviced requests per second, is measured. This is then repeated for different incoming request rate.

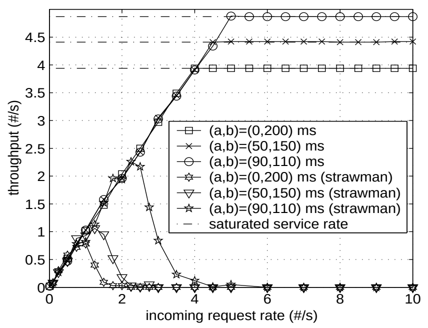

Figure 7. Throughput versus latency variance and contention, with latency uniformly distributed in (a, b) and average latency being 100ms. The 3-dashed lines correspond to theoretical predictions of saturated throughput, which differ with latency distribution. Data for both the strawman and Sigma are shown. *m/n* is 24/32.

Figure 7 depicts the throughput against different incoming request, varying the latency distribution. One can see that network latency distribution has little impact: the throughput increases linearly when request rate ramps up, until a point when it reaches the saturated rate and then stays flat as predicted by a theoretical model (see [14]). This is the ideal behavior. For comparison purposes, the throughput of the strawman protocol (Figure 3) is also plotted. We can see that the performance improvement is significant.

When replicas suffer from crash and thereafter undergo memory reset, performance will drop. There are many causes contributing to performance degradation and it's difficult to obtain a succinct reasoning. However, in a way a reset has the net effect of enlarging latency variance: a REQUEST, which would otherwise result in a successful RESPONSE, reaches the restarted replica behind those from others who should have been queued.

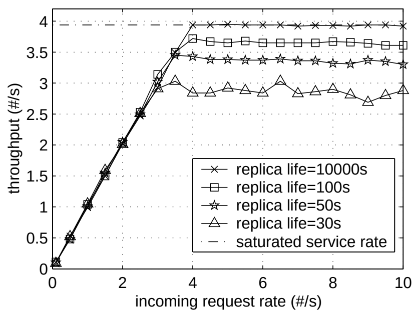

Figure 8. Throughput versus replica availability, with latency uniformly distributed in (0, 200)ms and *m/n* being 24/32. The dashed line corresponds to theoretical prediction of saturated throughput. When replica life is 10000 seconds, theoretical throughput can be achieved.

We set the average lifetime of replica to be excessively short. The lifetime is exponentially distributed [2] and different average values are tested. Figure 8 presents our results. The penalty of throughput is perceptible if the replica life is 30 seconds; however, it becomes less and less significant when replica life increases. Given that in a P2P environment, nodes typically will be online for about 10000 seconds [12], we believe that the ideal throughput of Sigma can be achieved in practice.

We have analyzed and measured message cost and show that it is asymptotically bound by 4*n*. Due to space limitation, we refer readers to the full technical report [14].

VI. Related Work

From the taxonomy of [10], the Sigma protocol would fall into the "permission-based" category. These protocols assume a closed system, in which clients are also the replicas. The context of this work mandates an open system where number of clients is unpredictable.

The more relevant work includes the Byzantine protocols [6][9][5] which also operates in an open-system setting. Obviously, Sigma's idea of virtual replicas is immediately applicable to these protocols to tailor-fit them in a P2P environment. The objectives, however, differ. Sigma is a light-weight synchronization protocol with O(n) message costs and does not attempt to deal with malicious client. Whereas the Byzantine protocols takes a replicated state machine approach with O(n^2) cost and handles malicious client. It is interesting to note that, for a total of 3f+1 replica, when faults exceed f, both protocols will yield unpredictable results.

Sigma's emphasis is more on the practicality side and pays much attention for performance. In the P2P space, [3] is similar to the strawman protocol, but is augmented with exponential backoff. It is not clear whether its property will hold in face of latency variance, which we believe is the prevailing pattern of a P2P environment.

VII. Conclusion and Future Work

The emerging P2P scenario brings forward several challenges to mutually exclusive access of the resource stored in it, such as the huge variance of network latency, unpredictable (and often very large) number of clients and finally high dynamism. These issues are partially addressed in previous works but not completely solved. In this paper, we propose the Sigma: a practical, efficient and fault-tolerant protocol for distributed mutual exclusion inside P2P DHT.

The key points of Sigma protocol are to use logical replicas and quorum consensus to deal with system dynamisms. Quasi-consistency and cooperation between clients and replicas circumvent the large variance of network latency and high contention. Sigma also gracefully deals with failure by two techniques: informed backoff and lease, making protocol fault-tolerant.

We verified that this protocol offers high performance in heterogeneous network condition and various contention rates. In a practical environment, the failure handling mechanism works well with negligible performance penalty and moderate communication overhead.

References

[1] B. Pawlowski, S. Shepler, et al. *The NFS Version 4 Protocol,* in Proceedings of the 2nd international system administration and networking conference (SANE2000)

[2] D. Liben-Nowell, H. Balakrishnan, and D. Karger, *Analysis of the Evolution of Peer-to-Peer Systems,* in 21st ACM Symposium on Principles of Distributed Computing (PODC), Monterey, CA, July 2002.

[3] G. Chokler, D. Malkhi, and M. Reiter, *Backoff Protocols for Distributed Mutual Exclusion and Ordering,* in Proceedings of 21st International Conference on Distributed Computing Systems (ICDCS), 2001.

[4] I. Stoica, et al, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications,* in Proceedings of ACM SIGCOMM 2001, San Deigo, CA, August 2001.

[5] J. Yin, et al. Separating Agreement from execution for Byzantine Fault Tolerant Services, in Proceedings of the 19th ACM Symposium on Operating Systems Principles, Octobor 2003.

[6] L. Lamport, R. Shostak and M. Pease, *The Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, 4(3):382-401, July 1982

[7] L. Lamport, *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM 21, 7 (July 1978), 558-565.

[8] M. K. Aguilera, W. Chen, and S. Toueg, *Failure detection and consensus in the crash-recovery model*. Distributed Computing, Springer-Verlag, 13:2, April 2000, pp. 99-125.

[9] M. Castro, B. Liskov, *Practical Byzantine Fault Tolerance*. in Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, February 1999.

[10] M. G. Velazquez, *A Survey of Distributed Mutual Exclusion Algorithms*, Colorado State University, Technical Report CS-93-116.

[11] N. Abramson, *The Aloha System – Another Alternative for Computer Communications*. In AFIPS Conference Proceedings, Vol. 36, 1970, pp. 295-298.

[12] S. Saroiu, P. Krishna Gummadi, S. D. Gribble, *A Measurement Study of Peer-to-Peer File Sharing Systems,* in Proceedings of Multimedia Computing and Networking (MMCN) 2002.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A Scalable Content-Addressable Network*, in Proceedings of ACM SIGCOMM 2001.

[14] S. Lin, et al, *A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems*, Microsoft Research, Technical Report.

[15] T. D. Chandra and S. Toueg, *Unreliable failure detectors for reliable distributed systems*. Journal of the ACM, 43(2):255-267.