A Real-Time Distributed Hash Table

Tao Qian, Frank Mueller
North Carolina State University, USA
mueller@cs.ncsu.edu

Yufeng Xin
RENCI, UNC-CH, USA
yxin@renci.org

***Abstract*—Currently, the North American power grid uses a centralized system to monitor and control wide-area power grid states. This centralized architecture is becoming a bottleneck as large numbers of wind and photo-voltaic (PV) generation sources require real-time monitoring and actuation to ensure sustained reliability. We have designed and implemented a distributed storage system, a real-time distributed hash table (DHT), to store and retrieve this monitoring data as a real-time service to an upper layer decentralized control system. Our real-time DHT utilizes the DHT algorithm Chord in a cyclic executive to schedule data-lookup jobs on distributed storage nodes. We formally define the pattern of the workload on our real-time DHT and use queuing theory to stochastically derive the time bound for response times of these lookup requests. We also define the quality of service (QoS) metrics of our real-time DHT as the probability that deadlines of requests can be met. We use the stochastic model to derive the QoS. An experimental evaluation on distributed nodes shows that our model is well suited to provide time bounds for requests following typical workload patterns and that a prioritized extension can increase the probability of meeting deadlines for subsequent requests.**

I. Introduction

The North American power grid uses Wide-Area Measure System (WAMS) technology to monitor and control the state of the power grid [17], [13]. WAMS increasingly relies on Phasor Measurement Units (PMUs), which are deployed to different geographic areas, e.g., the Eastern interconnection area, to collect real-time power monitoring data per area. These PMUs periodically send the monitored data to a phasor data concentrator (PDC) via proprietary Internet backbones. The PDC monitors and optionally controls the state of the power grid on the basis of the data. However, the current state-of-the-art monitoring architecture uses one centralized PDC to monitor all PMUs. As the number of PMUs is increasing extremely fast nowadays, the centralized PDC will soon become a bottleneck [5]. A straight-forward solution is to distribute multiple PDCs along with PMUs, where each PDC collects real-time data from only the part of PMUs that the PDC is in charge of. In this way, the PDC is not the bottleneck since the number of PMUs for each PDC could be limited and new PDCs could be deployed to manage new PMUs.

New problems arise with such a distributed PDC architecture. In today's power grid, real-time control actuation relies in part on the grid states of multiple areas [16], but with the new architecture the involved PMUs could be monitored by different PDCs. The first problem is how to manage the mapping between PDCs and PMUs so that a PDC can obtain PMU data from other PDCs. The second problem is how to communicate between these PDCs so that the real-time bounds on control operation are still guaranteed. For simplification, we consider these PDCs as distributed storage nodes over a wide-area network, where each of them periodically generates records as long as the PDC periodically collects data from the PMUs in that area. Then, the problem becomes how to obtain these records from the distributed storage system with real-time bounded response times.

Our idea is to build a distributed hash table (DHT) on these distributed storage nodes to solve the first problem. Similar to a single node hash table, a DHT provides *put(key, value)* and *get(key)* API services to upper layer applications. In our DHT, the data of one PMU are not only stored in the PDC that manages the PMU, but also in other PDCs that keep redundant copies according to the distribution resulting from the hash function and redundancy strategy. DHTs are well suited for this problem due to their superior scalability, reliability, and performance over centralized or even tree-based hierarchical approaches. The performance of the service provided by some DHT algorithms, e.g., Chord [15] and CAN [18], decreases only insignificantly while the number of the nodes in an overlay network increases. However, there is a lack of research on real-time bounds for these DHT algorithms for end-to-end response times of requests. Timing analysis is the key to solve the second problem. On the basis of analysis, we could provide statistical upper bounds for request times. Without loss of generality, we use the term *lookup request* or *request* to represent either a *put request* or a *get request*, since the core functionality of DHT algorithms is to look up the node that is responsible for storing data associated with a given key.

It is difficult to analyze the time to serve a request without prior information of the workloads on the nodes in the DHT overlay network. In our research, requests follow a certain pattern, which makes the analysis concrete. For example, PMUs periodically send real-time data to PDCs [13] so that PDCs issue put requests periodically. At the same time, PDCs need to fetch data from other PDCs to monitor global power states and control the state of the entire power grid periodically. Using these patterns of requests, we design a real-time model to describe the system. Our problem is motivated by power grid monitoring but our abstract real-time model provides a generic solution to analyze response times for real-time applications over networks. We further apply queuing theory to stochastically analyze the time bounds for requests. Stochastic approaches may break with traditional views of real-time systems. However, cyber-physical systems rely on stock Ethernet networks with enhanced TCP/IP so that soft real-time models for different QoS notions, such as simplex models [4], [1], are warranted.

Contributions: We have designed and implemented a real-time DHT by enhancing the Chord algorithm. Since a sequence of nodes on the Chord overlay network serve a lookup request, each node is required to execute one corresponding job. Our real-time DHT follows a cyclic executive [11], [2] to schedule these jobs on each node. We analyze the response time of these sub-request jobs on each node and aggregate them to bound the end-to-end response time for all requests. We also use a

This work was supported in part by NSF grants 1329780, 1239246, 0905181 and 0958311. Aranya Chakrabortty helped to scope the problem in discussions.

stochastic model to derive the probability that our real-time DHT can guarantee deadlines of requests. We issue a request pattern according to the needs of real-time distributed storage systems, e.g., power grid systems, on a cluster of workstations to evaluate our model. In general, we present a methodology for analyzing the real-time response time of requests served by multiple nodes on a network. This methodology includes: 1) employing real-time executives on nodes, 2) abstracting a pattern of requests, and 3) using a stochastic model to analyze response time bounds under the cyclic executive and the given request pattern. The real-time executive is not limited to a cyclic executive. For example, we show that a prioritized extension can increase the probability of meeting deadlines for requests that have failed during their first execution. This work, while originally motivated by novel control methods in the power grid, generically applies to distributed control of CPS real-time applications.

The rest of the paper is organized as follows. Section II presents the design and implementation details of our real-time DHT. Section III presents our timing analysis and quality of service model. Section IV presents the evaluation results. Section V presents the related work. Section VI presents the conclusion and the on-going part of our research.

II. Design and Implementation

Our real-time DHT uses the Chord algorithm to locate the node that stores a given data item. Let us first summarize the Chord algorithm and characterize the lookup request pattern (generically and specifically for PMU data requests). After that, we explain how our DHT implementation uses Chord and cyclic executives to serve these requests.

A. Chord

Chord provides storage and lookup capabilities of key/value pairs. Given a particular key, the Chord protocol locates the node that the key maps to. Chord uses consistent hashing to assign keys to Chord nodes organized as a ring network overlay [15], [7]. The consistent hash uses a base hash function, such as SHA-1, to generate an identifier for each node by hashing the node's IP address. When a lookup request is issued for a given key, the consistent hash uses the same hash function to encode the key. It then assigns the key to the first node on the ring whose identifier is equal to or follows the hash code of that key. This node is called the successor node of the key, or the target node of the key. Fig. 1 depicts an example of a Chord ring in the power grid context, which maps 9 PDCs onto the virtual nodes on the ring (labels in squares are their identifiers). As a result, PDCs issue requests to the Chord ring to periodically store and require PMU data from target PDCs. This solves the first problem that we have discussed in Section I.

In Chord, it is sufficient to locate the successor node of any given key by maintaining the nodes in a wrap-around circular list in which each node has a reference to its successor node. The lookup request is passed along the list until it encounters one node whose identifier is smaller than the hash code of the key but the identifier of the successor node is equal to or follows the hash code of the key. The successor of the node is the target node of the given key. Consider Fig. 1. Node $N4$ locates node $N15$ as the target node of $K14$ via intermediate

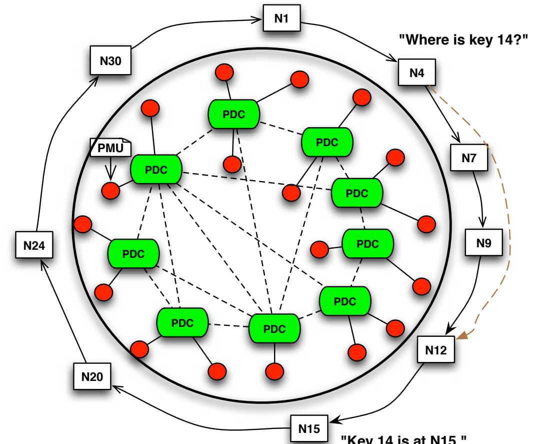

Fig. 1: PDC and Chord ring mapping example

TABLE I: Finger table for *N4*

#	start	interval	forward node
1	5	[5, 6)	N7
2	6	[6, 8)	N7
3	8	[8, 12)	N9
4	12	[12, 20)	N12
5	20	[20, 4)	N20

nodes *N7, N9, N12*. However, *O(N)* messages are required to find the successor node of a key, where N is the number of the nodes.

This linear algorithm is not scalable with increasing numbers of nodes. In order to reduce the number of intermediate nodes, Chord maintains a so-called finger table on each node, which acts as a soft routing table, to decide the next hop to forward lookup requests to. For example, assuming 5-bit numbers are used to represent node identifiers in Fig. 1, the finger table for *N4* is given in Table I. Each entry in the table indicates one routing rule: the next hop for a given key is the forward node in that entry if the entry interval includes the key. For example, the next hop for *K14* is *N12* as *14* is in *[12, 20)*. The interval for the 5th finger is [20, 36), which is [20, 4) as the virtual nodes are organized in a ring (modulo 32). In general, a finger table has $logN$ entries, where N is the number of bits of node identifiers. For node k, the start of the ith finger table entry is $k + 2^{i-1}$, the interval is $[k + 2^{i-1}, k + 2^{i})$, and the forward node is the successor of key $k + 2^{i-1}$.

To serve a lookup request for a given key, Chord first finds its predecessor, which is the first node in the counter-clockwise direction on the ring before the target node of that key, e.g., the predecessor of *K14* is *N12*. To find the predecessor, *N4* forwards *K14* to *N12* according to its finger table (shown as the dotted line in Fig. 1). Next, *N12* considers itself the predecessor of *K14* since *K14* is between *N12* and its successor *N15*. Chord returns the successor *N15* of the predecessor *N12* as the target node of *K14*. Each message forwarding reduces the distance to the target node to at least half that of the previous distance on the ring. Thus, the number of intermediate nodes per request is at most $logN$ with high probability [15], which is more scalable than the linear algorithm.

In addition to scalability, Chord tolerates high levels of churn in the distributed system. which makes it feasible to provide PMU data with the possibility that some PDC nodes or network links have failed in the power grid environment. However, resilience analysis with real-time considerations is beyond the scope of this paper. Thus, our model in this paper derives the response times for requests assuming that nodes do not send messages in the background for resilience purposes.

B. The pattern of requests and jobs

To eventually serve a request, a sequence of nodes are involved. The node that issues a request is called the initial node of the request, and other nodes are called subsequent nodes. In the previous example, *N4* is the initial node; *N12, N15* are subsequent nodes. As stated in Section I, requests in the power grid system follow a certain periodic pattern, which makes timing analysis feasible. In detail, each node in the system maintains a list of get and put tasks together with the period of each task. Each node releases periodic tasks in the list to issue lookup requests. Once a request is issued, one job on each node on the way is required to be executed until the request is eventually served.

Jobs on the initial nodes are periodic, since they are initiated by periodic real-time tasks. However, jobs on the subsequent nodes are not necessarily periodic, since they are driven by the lookup messages sent by other nodes via the network. The network delay to transmit packets is not always constant, even through one packet is enough to forward a request message as the length of PMU data is small in the power grid system. Thus, the pattern of these subsequent jobs depends on the node-to-node network delay. With different conditions of the network delay, we consider two models.

(1) In the first model, the network delay to transmit a lookup message is constant. Under this assumption, we use periodic tasks to handle messages sent by other nodes. For example, assume node A has a periodic PUT task $\tau_1(0, T_1, C_1, D_1)$. We use the notation $\tau(\phi, T, C, D)$ to specify a periodic task τ, where ϕ is its phase, T is the period, C is the worst case execution time per job, and D is the relative deadline. From the schedule table for the cyclic executive, the response times for the jobs of τ_1 in a hyperperiod H are known [11]. Assume two jobs of this task, $J_{1,1}$ and $J_{1,2}$, are in one hyperperiod on node A, with $R_{1,1}$ and $R_{1,2}$ as their release times relative to the beginning of the hyperperiod and $W_{1,1}$ and $W_{1,2}$ as the response times, respectively.

Let the network latency, δ, be constant. The subsequent message of the job $J_{1,1}$ is sent to node B at time $R_{1,1}+W_{1,1}+\delta$. For a sequence of hyperperiods on node A, the jobs to handle these subsequent messages on node B become a periodic task $\tau_2(R_{1,1} + W_{1,1} + \delta, H, C_2, D_2)$. The period of the new task is H as one message of $J_{1,1}$ is issued per hyperperiod. Similarly, jobs to handle the messages of $J_{1,2}$ become another periodic task $\tau_3(R_{1,2} + W_{1,2} + \delta, H, C_2, D_2)$ on node B.

(2) In the second model, network delay may be variable, and its distribution can be measured. We use aperiodic jobs to handle subsequent messages. Our DHT employs a cyclic executive on each node to schedule the initial periodic jobs and subsequent aperiodic jobs, as discussed in the next section.

We focus on the second model in this paper since this model reflects the networking properties of IP-based network infrastructure. This is also consistent with current trends in cyber-physical systems to utilize stock Ethernet networks with TCP/IP enhanced by soft real-time models for different QoS notions, such as the simplex model [4], [1]. The purely periodic model is subject to future work as it requires special hardware and protocols, such as static routing and TDMA to avoid congestion, to guarantee constant network delay.

C. Job scheduling

Each node in our system employs a cyclic executive to schedule jobs using a single thread. The schedule table $L(k)$ indicates which periodic jobs to execute at a specific frame k in a hyperperiod of F frames. L is calculated offline from periodic task parameters [11]. Each node has a FIFO queue of released aperiodic jobs. The cyclic executive first executes periodic jobs in the current frame. It then executes the aperiodic jobs (up to a given maximum time allotment for aperiodic activity, similar to an aperiodic server). If the cyclic executive finishes all the aperiodic jobs in the queue before the end of the current frame, it waits for the next timer interrupt. Algorithm 1 depicts the work of the cyclic executive, where f is the interval of each frame.

```
Data: L, aperiodic job queue Q, current job
1  Procedure SCHEDULE
2      current ← 0
3      setup timer to interrupt every f time, execute
       TIMER_HANDLER when it interrupts
4      while true do
5          current job ← L(current)
6          current = (current + 1)%F
7          if current job ≠ nil then
8              execute current job
9              mark current job done
10         end
11         while Q is not empty do
12             current job ← remove the head of Q
13             execute current job
14             mark current job done
15         end
16         current job ← nil
17         wait for next timer interrupt
18     end
19 Procedure TIMER_HANDLER
20     if current job ≠ nil and not done then
21         if current job is periodic then
22             mark the current job failed
23         else
24             save state and push current job back to the
               head of Q
25         end
26     end
27     jump to Line 4
```

Algorithm 1: Pseudocode for the cyclic executive

In addition to jobs that handle request messages, our DHT uses a task to receive messages from other nodes. The cyclic executive schedules this receiving task as just another periodic task. This receiving task periodically moves the messages from the buffer of the underlying network stack to the aperiodic job queue. This design avoids the cost of context switches between the cyclic executive and another thread to receive messages, which could interrupt the cyclic executive at a random time whenever a message arrives. As a side effect, messages can only be scheduled in the next frame affects the periodic receiving jobs executing. This increases the response times of aperiodic jobs. Since the frame size is small (in the level of milliseconds), this delay is acceptable for our application domain.

TABLE II: Types of Messages passed among nodes

Type [1]	**Parameters** [2]	**Description**
PUT	:key:value	the initial put request to store (key:value).
PUT_DIRECT	:ip:port:sid:key:value	one node sends to the target node to store (key:value). [3]
PUT_DONE	:sid:ip:port	the target node notifies sender of successfully handling the PUT_DIRECT.
GET	:key	the initial get request to get the value for the given key.
GET_DIRECT	:ip:port:sid:key	one node sends to the target node to get the value.
GET_DONE	:sid:ip:port:value	the target node sends the value back as the feed back of the GET_DIRECT.
GET_FAILED	:sid:ip:port	the target node has no associated value.
LOOKUP	:hashcode:ip:port:sid	the sub-lookup message. [4]
DESTIN	:hashcode:ip:port:sid	similar to LOOKUP, but DESTIN message sends to the target node.
LOOKUP_DONE	:sid:ip:port	the target node sends its address back to the initial node.

[1] Message types for Chord *fix_finger* and *stabilize* operations are omitted.
[2] One message passed among the nodes consists of its type and the parameters, e.g, PUT:PMU-001:15.
[3] *(ip:port)* is the address of the node that sends the message. Receiver uses it to locate the sender. *sid* is unique identifier.
[4] Nodes use finger tables to determine the node to pass the LOOKUP to. *(ip:port)* is the address of the request initial node.

D. A Real-time DHT

Our real-time DHT is a combination of Chord and a cyclic executive to provide predictable response times for requests following the request pattern. Our DHT adopts Chord's algorithm to locate the target node for a given key. On the basis of Chord, our DHT provides two operations: *get(key)* and *put(key, value)*, which obtains the paired value with a given key and puts one pair onto a target node, respectively. Table II describes the types of messages exchanged among nodes. Algorithm 2 depicts the actions for a node to handle these messages. We have omitted the details of failure recovery messages such as *fix_finger* and *stabilize* in Chord, also implemented as messages in our DHT. $node.operation(parameters)$ indicates that the operation with given parameters is executed on a the remote node, implemented by sending a corresponding message to this node.

Example: In order to serve the periodic task to *get K14* on node *N4* in Fig. 1, the cyclic executive on *N4* schedules a periodic job *GET* periodically, which sends a *LOOKUP* message to *N12*. *N12* then sends a *DESTIN* message to *N15* since it determines that *N15* is the target node for *K14*. *N15* sends *LOOKUP_DONE* back to the initial node *N4*. Our DHT stores the request detail in a buffer on the initial node and uses a unique identifier (sid) embedded in messages to identify this request. When *N4* receives the *LOOKUP_DONE* message from the target node, it releases an aperiodic job to obtain the request detail from the buffer and continues by sending *GET_DIRECT* to the target node *N15* as depicted in the pseudocode. *N15* returns the value to *N4* by sending a *GET_DONE* message back to *N4*. Now, *N4* removes the request detail from the buffer and the request is completed.

III. Analysis

As a decentralized distributed storage system, the nodes in our DHT work independently. In this section, we first explain the model used to analyze the response times of jobs on a single node. Then, we aggregate these times to bound the end-to-end time for all requests.

A. Job response time on a single node

Let us first state an example of job patterns on a single node to illustrate the problem. Let us assume the hyperperiod of the periodic requests is $30ms$, and the periodic receiving task is executed three times in one hyperperiod at $0ms$ (the beginning of the hyperperiod), $10ms$, and $20ms$, respectively. The timeline of one hyperperiod can be divided into three slots according to the period of the receiving task. In every slot, the cyclic executive utilizes the first 40% of its time to execute periodic jobs, and the remaining time to execute aperiodic jobs as long as the aperiodic job queue is not empty. Since aperiodic jobs are released when the node receives messages from other nodes, we model the release pattern of aperiodic jobs as a homogeneous Poisson process with $2ms$ as the average inter-arrival time. The execution time is $0.4ms$ for all aperiodic jobs. The problem is to analyze response times of these aperiodic jobs. The response time of an aperiodic job in our model consists of the time waiting for the receiving task to move its message to the aperiodic queue, the time waiting for the executive to initiate the job, and the execution time of the job.

An M/D/1 queuing model [8] is suited to analyze the response times of aperiodic jobs if these aperiodic jobs are executed once the executive has finished all periodic jobs in that time slot. In our model, the aperiodic jobs that arrive at this node in one time slot could only be scheduled during the next time slot, as these aperiodic jobs are put into the job queue only when the receiving task is executed at the beginning of the next time slot. The generic M/D/1 queuing model cannot capture this property. We need to derive a modified model.

Formally, Table III includes the notation we use to describe our model. We also use the same notation without the subscript for the vector of all values. For example, U is $(U_0, U_1, \ldots, U_K)$, C is $(C_0, C_1, \ldots, C_M)$. In the table, C is obtained by measurement from the implementation; H, K, ν, U are known from the schedule table; M is defined in our DHT algorithm. We explain λ in detail in Section III-B.

Given a time interval of length x and arrivals g in that interval, the total execution time of aperiodic jobs $E(x, g)$ can be calculated with Equation 1. Without loss of generality, we use notation E and $E(x)$ to represent $E(x, g)$.

$$E(x, g) = \sum_{i=1}^{M} C_i * g_i \; with \; probability \prod_{i=1}^{M} P(g_i, \lambda_i, x) \quad (1)$$

We further define A_p as the time available to execute aperiodic jobs in time interval (ν_0, ν_{p+1}).

$$A_p = \sum_{i=0}^{p} (1 - U_i) * (\nu_{i+1} - \nu_i), 0 \le p \le K. \quad (2)$$

However, the executive runs aperiodic jobs only after periodic jobs have finished in a frame. We define $W(i, E)$ as the response time for an aperiodic job workload E if these jobs are scheduled after the ith receiving job. For $1 \le i \le K$,

```
1  Procedure GET(key) and PUT(key, value)
2      sid ← unique request identifier
3      code ← hashcode(key)
4      address ← ip and port of this node
5      store the request in the buffer
6      execute LOOKUP(address, sid, code)
7  Procedure LOOKUP(initial-node, sid, hashcode)
8      next ← next node for the lookup in the finger table
9      if next = target node then
10         next.DESTIN(initial-node, sid, hashcode)
11     else
12         next.LOOKUP(initial-node, sid, hashcode)
13     end
14 Procedure DESTIN(initial-node, sid, hashcode)
15     address ← ip and port of this node
16     initial-node.LOOKUP_DONE(address, sid)
17 Procedure LOOKUP_DONE(target-node, sid)
18     request ← find buffered request using sid
19     if request = get then
20         target-node.GET_DIRECT(initial-node, sid,
           request.key)
21     else if request = put then
22         target-node.PUT_DIRECT(initial-node, sid,
           request.key, request.value)
23     else if request = fix finger then
24         update the finger table
25     end
26 Procedure GET_DIRECT(initial-node, sid, key)
27     address ← ip and port of this node
28     value ← find value for key in local storage
29     if value = nil then
30         initial-node.GET_FAILED(address, sid)
31     else
32         initial-node.GET_DONE(address, sid, value)
33     end
34 Procedure PUT_DIRECT(initial-node, sid, key, value)
35     address ← ip and port of this node
36     store the pair to local storage
37     initial-node.PUT_DONE (address, sid)
38 Procedure PUT_DONE, GET_DONE, GET_FAILED
39     execute the associated operation
```

Algorithm 2: Pseudocode for message handling jobs

$W(i, E)$ is a function of E calculated from the schedule table in three cases:

(1) If $E \leq A_i - A_{i-1}$, which means the executive can use the aperiodic job quota between $(\nu_i, \nu_{i+1}]$ to finish the workload, we can use the parameters of the periodic job schedule in the schedule table to calculate $W(i, E)$.

(2) If $\exists p \in [i+1, K]$ so that $E \in (A_{p-1} - A_{i-1}, A_p - A_{i-1}]$, which means the executive utilizes all the aperiodic job quota between (ν_i, ν_p) to execute the workload and finishes the workload at a time between (ν_p, ν_{p+1}), then $W(i, E) = \nu_p - \nu_i + W(p, E - A_{p-1} + A_{i-1})$.

(3) In the last case, the workload is finished in the next hyperperiod. $W(i, E)$ becomes $H - \nu_i + W(0, E - A_K + A_{i-1})$. $W(0, E)$ indicates that one can use the aperiodic quota before the first receiving job to execute the workload. If the first receiving job is scheduled at the beginning of the hyperperiod, this value is the same as $W(1, E)$. In addition, we require that any workload finishes before the end of the next hyperperiod. This is accomplished by analyzing the timing of receiving jobs and ensures that the aperiodic job queue is in the same state at the beginning of each hyperperiod, i.e., no workload accumulates from previous hyperperiods (except for the last hyperperiod).

TABLE III: Notation

Notation	Meaning
H	hyperperiod
K	number of receiving jobs in one hyperperiod
ν_i	time when the ith receiving job is scheduled [1,2]
U_i	utilization of periodic jobs in time interval (ν_i, ν_{i+1})
M	number of different types of aperiodic jobs
λ_i	average arrival rate of the ith type aperiodic jobs
C_i	worst case execution time of the ith type aperiodic jobs
g_i	number of arrivals of ith aperiodic job
$E(x, g)$	total execution time for aperiodic jobs that arrive in time interval of length x
$W(i, E)$	response time for aperiodic jobs if they are scheduled after i^{th} receiving job
$P(n, \lambda, x)$	probability of n arrivals in time interval of length x, when arrival is a Poisson process with rate λ

[1] ν_i are relative to the beginning of the current hyperperiod.
[2] For convenience, ν_0 is defined as the beginning of a hyperperiod, ν_{K+1} is defined as the beginning of next hyperperiod.

Let us assume an aperiodic job J of execution time C_m arrives at time t relative to the beginning of the current hyperperiod. Let $p+1$ be the index of the receiving job such that $t \in [\nu_p, \nu_{p+1})$. We also assume that any aperiodic job that arrives in this period is put into the aperiodic job queue by this receiving job. Then, we derive the response time of this job in different cases.

(1) The periodic jobs that are left over from the previous hyperperiod and arrive before ν_p in the current frame cannot be finished before ν_{p+1}. Equation 3 is the formal condition for this case, in which HLW is the leftover workload from the previous hyperperiod.

$$HLW + E(\nu_p) - A_p > 0 \tag{3}$$

In this case, the executive has to first finish this leftover workload, then any aperiodic jobs that arrive in the time period $[\nu_p, t)$, which is $E(t - \nu_p)$, before executing job J. As a result, the total response time of job J is the time to wait for the next receiving job at ν_{p+1}, which puts J into the aperiodic queue, and the time to execute aperiodic the job workload $LW(\nu_p) + E(t - \nu_p) + C_m$, which is $W(p+1, HLW + E(t) - A_p + C_m)$, after the $(p+1)^{th}$ receiving job. The response time of J is expressed in Equation 4.

$$R(C_m, t) = \begin{array}{l}(\nu_{p+1} - t) + \\ W(p+1, HLW + E(t) - A_p + C_m)\end{array} \tag{4}$$

(2) In the second case, the periodic jobs that are left over from the previous hyperperiod and arrive before ν_p can be finished before ν_{p+1}; the formal condition and the response time are given by Equations 5 and 6, respectively.

$$HLW + E(\nu_p) \leq A_p \tag{5}$$

$$R(C_m, t) = (\nu_{p+1} - t) + W(p+1, E(t - \nu_p) + C_m) \tag{6}$$

The hyperperiod leftover workload HLW is modeled as follows. Consider three continuous hyperperiods, H_1, H_2, H_3. The leftover workload from H_2 consists of two parts. The

first part, $E(H - \nu_K)$, is comprised of the aperiodic jobs that arrive after the last receiving job in H_2, as these jobs can only be scheduled by the first receiving job in H_3. The second part, $E(H + \nu_K) - 2A_{K+1}$, is the jobs that arrive in H_1 and before the last receiving job in H_2. These jobs have not been scheduled during the entire aperiodic allotment over H_1 and H_2. We construct a distribution for HLW in this way. In addition, we can construct a series of distributions of HLW, where sequences of hyperperiods with different lengths are considered. This series converges (after three hyperperiods) to a distribution subsequently used as the overall HLW distribution for this workload. However, our evaluation shows that two previous hyperperiods are sufficient for the request patterns.

This provides the stochastic model $R(C_m, t)$ for the response time of an aperiodic job of execution time C_m that arrives at a specific time t. By sampling t in one hyperperiod, we have obtained the stochastic model for the response time of aperiodic jobs that arrive at any time.

B. End-to-end response time analysis

We aggregate the single node response times and network delays to transmit messages for the end-to-end response time of requests. The response time of any request consists of four parts: (1) the response time of the initial periodic job — this value is known by the schedule table; (2) the total response time of jobs to handle subsequent lookup messages on at most $logN$ nodes with high probability [15], where N is the number of nodes; (3) the response time of aperiodic jobs to handle LOOKUP_DONE, and the final pair of messages, e.g., PUT_DIRECT and PUT_DONE; (4) the total network delays to transmit these messages. We use a value δ based on measurements for the network delay, where $P(network\ delay \leq \delta) \geq T$, for a given threshold T.

To use the single node response time model, we need to know the values of the model parameters of Table III. With the above details on requests, we can obtain H, K, v, U from the schedule table. λ is defined as follows: let T be the period of the initial request on each node, then $\frac{N}{T}$ new requests arrive at our DHT in one time unit, and each request issues at most $logN$ subsequent lookup messages. Let us assume that hash codes of nodes and keys are randomly located on the Chord ring, which is of high probability with the SHA-1 hashing algorithm. Then each node receives $\frac{logN}{T}$ lookup messages in one time unit. The arrival rate of LOOKUP and DESTIN messages is $\frac{logN}{T}$. In addition, each request eventually generates one LOOKUP_DONE message and one final pair of messages these messages, the arrive rate is $\frac{1}{T}$.

C. Quality of service

We define quality of service (QoS) as the probability that our real-time DHT can guarantee requests to be finished before their deadlines. Formally, given the relative deadline D of a request, we use the stochastic model $R(C_m, t)$ for single node response times and the aggregation model for end-to-end response times to derive the probability that the end-to-end response time of the request is equal to or less than D. In this section, we apply the formula of our model step by step to explain how to derive this probability in practice.

The probability density function $\rho(d, C_m)$ is defined as the probability that the single node response time of a job with execution time C_m is d. We first derive the conditional density function $\rho(d, C_m|t)$, which is the probability under the condition that the job arrives at time t, i.e., the probability that $R(C_m, t) = d$. Then, we apply the law of total probability to derive $\rho(d, C_m)$. The conditional density function $\rho(d, C_m|t)$ is represented as a table of pairs $\pi(\rho, d)$, where ρ is the probability that an aperiodic job finishes with response time d. We apply the algorithms described in Section III-A to build this table as follows. Let $p + 1$ be the index of the receiving job such that $t \in [\nu_p, \nu_{p+1})$.

(1) We need to know when to apply Equations 4 and 6. This is determined by the probability χ that condition $HLW + E(\nu_p, g) \leq A_p$ holds. To calculate χ, we enumerate job arrival vectors g that have significant probabilities in time $(0, \nu_p)$ according to the Poisson distribution, and use Equation 1 to calculate the workload $E(\nu_p, g)$ for each g. The term significant probability means any probability that is larger than a given threshold, e.g., 0.0001%. Since the values of HLW and $E(\nu_p, g)$ are independent, the probability of a specific pair of HLW and arrival vector g is given by simply their product. As a result, we build a condition table $CT(g, \rho)$, in which each row represents a pair of vector g, which consists of the numbers of aperiodic job arrivals in time interval $(0, \nu_p)$ under the condition $HLW + E(\nu_p, g) \leq A_p$, and the corresponding probability ρ for that arrival vector. Then, $\chi = \sum \rho$ is the total probability for that condition.

(2) We construct probability density table $\pi_2(\rho, d)$ for response times of aperiodic jobs under condition $HLW + E(\nu_p) \leq A_p$. In this case, we enumerate job arrival vectors g that have significant probabilities in time $(0, t - \nu_p)$ according to the Poisson distribution. We use Equation 1 to calculate their workload and Equation 6 to calculate their response time for each g. Each job arrival vector generates one row in density table π_2.

(3) We construct probability density table $\pi_3(\rho, d)$ for response times of aperiodic jobs under condition $HLW + E(\nu_p) > A_p$. We enumerate job arrival vectors g that have significant probabilities in time $(0, t)$ according to the Poisson distribution. We use Equation 4 to calculate response times. Since time interval $(0, t)$ includes $(0, \nu_p)$, arrival vectors that are in condition table CT must be excluded from π_3, because rows in CT only represent arrivals that results in $HLW + E(\nu_p) \leq A_p$. We normalize the probabilities of the remaining rows. By normalizing, we mean multiplying each probability by a common constant factor, so that the sum of the probabilities is 1.

(4) We merge the rows in these two density tables to build the final table for the conditional density function. Before merging, we need to multiply every probability in table π_2 by the weight χ, which indicates the probability that rows in table π_2 are valid. For the same reason, every probability in table π_3 is multiplied by $(1 - \chi)$.

Now, we apply the law of total probability to derive $\rho(d, C_m)$ from the conditional density functions by sampling t in $[0, H)$. The conditional density tables for all samples are merged into a final density table $\prod(\rho, d)$. The samples are uniformly distributed in $[0, H)$ so that conditional density tables have the same weight during the merge process. After normalization, table $\prod$ is the single node response time density function $\rho(d, C_m)$.

We apply the aggregation rule described in Section III-B to derive the end-to-end response time density function on the

```
1  Procedure UNIQUE (π)
2      π_des ← empty table
3      for each row (ρ, d) in π do
4          if row (ρ_old, d) exists in π_des then
5              ρ_old ← ρ_old + ρ
6          else
7              add row (ρ, d) to π_des
8          end
9      end
10     return π_des
11 Procedure SUM (π1, π2)
12     π3 ← empty table
13     for each row (ρ1, d1) in π1 do
14         for each row (ρ2, d2) in π2 do
15             add row (ρ1 * ρ2, d1 + d2) to π3
16         end
17     end
18     return UNIQUE(π3)
```

Algorithm 3: Density tables operations

basis of $\rho(d, C_m)$. According to the rule, end-to-end response time includes the response time of the initial periodic job, network delays, and the total response time of $(logN + 3)$ aperiodic jobs of different types. In order to represent the density function of the total response time for these aperiodic jobs, we define the operation SUM on two density tables π_1 and π_2 as in Algorithm 3. The resulting density table has one row $(\rho_1{*}\rho_2, d_1{+}d_2)$ for each pair of rows (ρ_1, d_1) and (ρ_2, d_2) from table π_1 and π_2, respectively. That is, each row in the result represents one sum of two response times from the two tables and the probability of the aggregated response times. The density function of the total response time for $(logN+3)$ aperiodic jobs is calculated as Equation 7 (SUM on all π_i), where π_i is the density function for the ith job.

$$\rho(d) = \sum_{i=1}^{logN+3} \pi_i \tag{7}$$

The maximum number of rows in density table $\rho(d)$ is $2(logN + 3)H\omega$, where $2H$ is the maximum response time of single node jobs, and ω is the sample rate for arrival time t that we use to calculate each density table.

Let us return to the QoS metric, i.e., the probability that a request can be served within a given deadline D. We first reduce D by the fixed value Δ, which includes the response time for the initial periodic job and network delays $(logN + 3)\delta$. Then, we aggregate rows in the density function $\rho(d)$ to calculate this probability $P(D - \Delta)$.

$$P(D - \Delta) = \sum_{(\rho_i, d_i) \in \rho(d), d_i \leq D - \Delta} \rho_i \tag{8}$$

IV. Evaluation

We evaluated our real-time DHT on a local cluster with 2000 cores over 120 nodes. Each node features a 2-way SMP with AMD Opteron 6128 (Magny Core) processors and 8 cores per socket (16 cores per node). Each node has 32GB DRAM and Gigabit Ethernet (utilized in this study) as well as Infiniband Interconnect (not used here). We apply workloads of different intensity according to the needs of the power grid control system on different numbers of nodes (16 nodes are utilized in our experiments), which act like PDCs. The nodes are not synchronized to each other relative to their start of hyperperiods as such synchronization would be hard to maintain in a distributed system. We design experiments for different intensity of workloads and then collect single-node and end-to-end response times of requests in each experiment. The intensity of a workload is quantified by the system utilization under that workload. The utilization is determined by the number of periodic lookup requests and other periodic power control related computations. The lookup keys have less effect on the workload and statistic results as long as they are evenly stored on the nodes, which has high probability in Chord. In addition, the utilization of aperiodic jobs is determined by the number of nodes in the system. The number of messages passing between nodes increases logarithmically with the number of nodes, which results in an increase in the total number of aperiodic jobs. We compare the experimental results with the results given by our stochastic model for each workload.

In the third part of this section, we give experimental results of our extended real-time DHT, in which the cyclic executive schedules aperiodic jobs based on the priorities of requests. The results show that most of the requests that have tight deadlines can be finished at the second trial under the condition that the executive did not finish the request the first time around.

A. Low workload

In this experiment, we implement workloads of low utilizations as a cyclic executive schedule. A hyperperiod (30ms) contains three frames of 10ms, each with a periodic followed by an aperiodic set of jobs. The periodic jobs include the receiving job for each frame and the following put/get jobs: In frame 1, each node schedules a *put* request followed by a *get* request; in frame 2 and 3, each node schedules a *put* request, respectively. In each frame, the cyclic executive schedules a *compute* job once the periodic jobs in that frame have been executed. This *compute* job is to simulate periodic computations on PDCs for power state estimation, which is implemented as a tight loop of computation in our experiments. As a result, the utilizations of periodic jobs in the three frames are all 40%. Any put/get requests forwarded to other nodes in the DHT result in aperiodic (remote) jobs. The execution time of aperiodic jobs is $0.4ms$. The system utilization is 66.7% (40% for periodic jobs and 26.7% for aperiodic jobs) when the workload is run with 4 DHT nodes.

Results are plotted over 3,000 hyperperiods. Fig. 2 depicts the single-node response times. The red dots forming a cloud-like area in the figure are measured response times of jobs. Since our stochastic model derives a distribution of response times for jobs arriving at every time instance relative to the hyperperiod, we depict the mathematical expectation and the maximum value for each time instance, which are depicted as blue (in middle of the red clouds) and black (top) lines in the figure, respectively. The figure shows that messages sent at the beginning of each frame experience longer delays before they are handled by corresponding aperiodic jobs with proportionally higher response times. The reason for this is that these messages spend more time waiting for the execution of the next receiving job so that their aperiodic jobs can be scheduled. The modeled average times (blue/middle of

clouds) follow a proportionally decaying curve delimited by the respective periodic workload of a frame (4ms) plus the frame size (10ms) as the upper bound (left side) and just the periodic workload as the lower bound (right side). The figure shows that the measured times (red/clouds) closely match this curve as the average response time for most of the arrival times.

Maximum modeled times have a more complex pattern. In the first frame, their response times are bounded by 18ms for the first 4ms followed by a nearly proportionally decaying curve (22ms-18ms response time) over the course of the next 6ms. The spike at 4ms is due to servicing requests that arrived in the first 4ms, including those from the previous hyperperiod. Similar spikes between frames exist for the same reason, where their magnitude is given by the density of remaining aperiodic jobs and the length of the periodic workload, which also accounts for the near-proportional decay. This results in aperiodic jobs waiting two frames before they execute when issued during the second part of each frame.

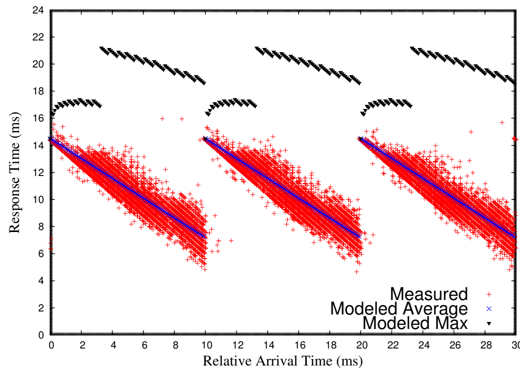

Fig. 2: Low-workload single-node response times (4 nodes)

Fig. 3 depicts the cumulative distributions of single-node response times for aperiodic jobs. The figure shows that 99.3% of the aperiodic jobs finish within the next frame after they are released under this workload, i.e., their response times are bounded by $14.4ms$. Our model predicts that 97.8% of aperiodic jobs finish within the $14.4ms$ deadline, which is a good match. In addition, for most of the response times in the figure, our model predicts that a smaller fraction of aperiodic jobs finish within the response times than the fraction in the experimental data as the blue (lower/solid) curve for modeled response times is below the red (upper/dashed) curve, i.e., the former effectively provides a lower bound for the latter. This indicates that our model is conservative for low workloads.

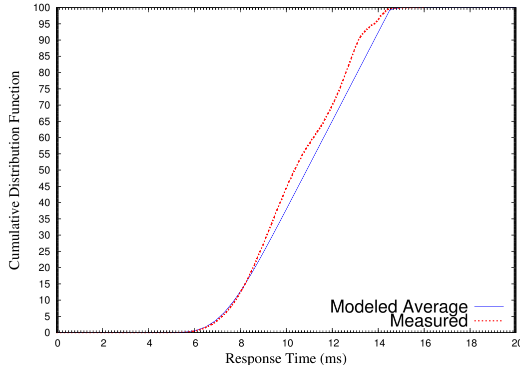

Fig. 3: Low-workload single-node resp. times distr. (4 nodes)

Fig. 4 depicts the cumulative distributions of end-to-end response times for requests (left/right two lines for 4/8 nodes), i.e., the time between a put/get request and its final response

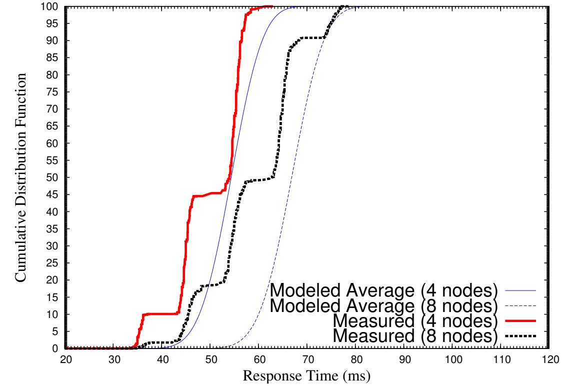

Fig. 4: Low-workload end-to-end response times distribution

after propagating over multiple hops (nodes) within the DHT. This figure shows that measured response times for four nodes have three clusters centered around $35ms$, $45ms$, and $55ms$. Every cluster has a base response time (about $35ms$) due to the LOOK_DONE and the final pair of messages as discussed in Section III-B. In addition, different clusters indicate that requests contact different numbers of nodes to eventually serve the requests. One more intermediate node contacted increases the response time by around $10ms$, which is the average single-node response time as depicted in Fig. 3. Approximately 10% of the requests are served without an intermediate node. This happens when the initial node of a request is also the target node. In this case, our current DHT implementation sends a LOOKUP_DONE message directly to itself. Our QoS model considers the worst case where $logN$ nodes are required, which provides a probabilistic upper bound on response times. In addition, this figure shows that our QoS model is conservative. For example, 99.9% of the requests are served within $62ms$ in the experiment, while our model indicates a 93.9% coverage for that response time.

In addition, we evaluate our model with the same request pattern with 8 DHT nodes. The system utilization becomes 72% as the utilization for aperiodic jobs increases because of the additional nodes in the DHT. Fig. 4 (right-most lines) depicts the cumulative distributions of end-to-end response times for requests. Four clusters of response times are shown in this figure for our measurements is the black/right stepped curve. Here, the base value is larger between $40ms$ and $45ms$ as the utilization has increased. One more intermediate node is contacted for a fraction of requests compared to the previous experiment with 4 DHT nodes.

B. *High workload*

In this experiment, we increase the utilization of the workloads for the cyclic executive. A hyperperiod (40ms) contains four frames of 10ms, each with a periodic followed by an aperiodic set of jobs. The periodic jobs include a receiving job and two put/get jobs for each frame. Any put/get requests forwarded to other nodes in the DHT result in aperiodic jobs. The execution time of aperiodic jobs are $0.4ms$. The system utilization is 88.0% when the workload is run with 8 DHT nodes, which includes 40% for periodic jobs and 44.0% for aperiodic jobs.

Fig. 5 depicts the single-node response times. The red dots/cloud in the figure depict measured response times of jobs. The blue/within the cloud and black/top lines are the mathematical expectation and maximum response times for

each sample instance given by our model, respectively. Compared with Fig. 2, a part of the red/cloud area at the end of each frame, e.g., between $8ms$ to $10ms$ for the first frame, moves up indicating response times larger than $14.4ms$. This is due to a larger fraction of aperiodic jobs during these time intervals that are scheduled after the next frame (i.e., in the third frame) due to the increase of system utilization. Our model also shows this trend as the tails of blue lines curve up slightly. Figures 6 and 7 depict the cumulative distributions of single-node and end-to-end response times under this workload, respectively. Compared with Figures 3 and 4, the curves for higher utilization move to the right. This suggests larger response times. Our model provides an upper bound on response times of 99.9% of all requests. Fig. 7 (right-side/dotted lines) also depicts the cumulative distribution of end-to-end response times of requests running on 16 DHT nodes. Our model also provides reliable results for this case as the curves for the experimental data are slightly above the curves of our model.

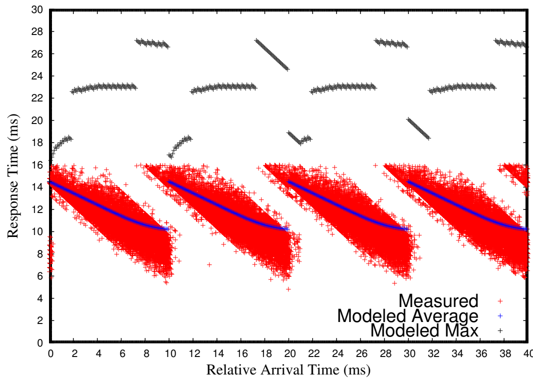

Fig. 5: High-workload single-node response times (8 nodes)

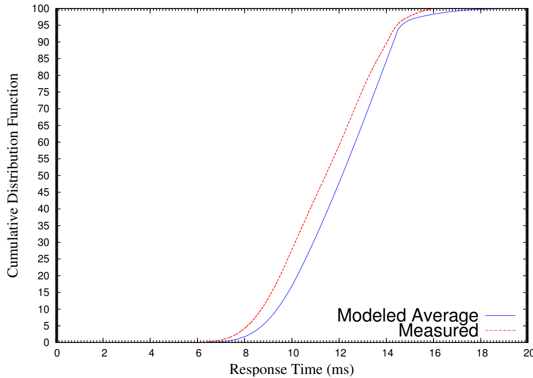

Fig. 6: High-workload single-node resp. times distr. (8 nodes)

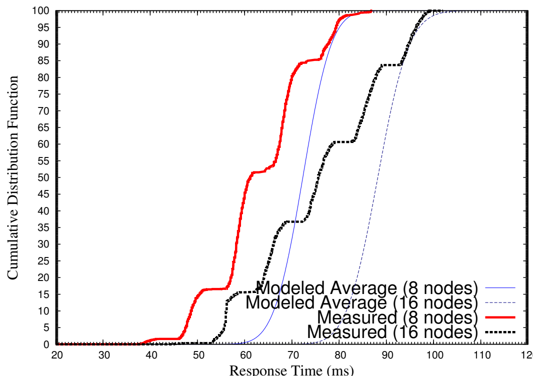

Fig. 7: High-workload end-to-end response times distribution

These experiments demonstrate that a stochastic model can result in highly reliable response times bounds, where the probability of timely completion can be treated as a quality of service (QoS) property under our model.

C. Prioritized queue extension

We extend our real-time DHT to use prioritized queues to schedule aperiodic jobs. Our current stochastic model does not take prioritized queues into consideration, so we compare the results with that of the real-time DHT without prioritized queue for the low workload executed on 4 DHT nodes.

In this extension, the system assigns the lowest priority to jobs of requests that are released for the first time and sets a timeout at their deadline. If the system fails to complete a request before its deadline, it increases the priority of its next job release until that request is served before its deadline. To implement this, all the messages for a request inherit the maximum remaining response time, which is initially the relative deadline, to indicate the time left for that request to complete. This time is reduced by the single node response time (when the aperiodic job for that request is finished) plus the network delay δ if a request is forwarded to the next node. When a node receives a message that has no remaining response time, the node simply drops the message instead of putting it onto aperiodic job queues. Cyclic executives always schedule aperiodic jobs with higher priority first (FCFS for the jobs of same priority).

In the experiment, we set the relative deadline of requests to $55ms$, which is the duration within which half of the requests with two intermediate nodes can be served.

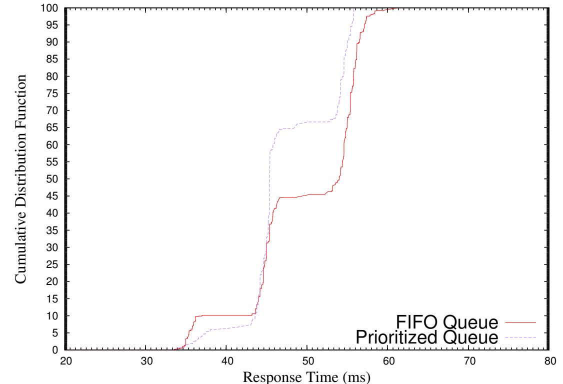

Fig. 8: End-to-end response times distribution comparison

The purple/left curve in Fig. 8 depicts the cumulative distribution of end-to-end response times of requests served by the prioritized implementation. Requests that require three intermediate nodes have higher probability to miss the $55ms$ relative deadline at first. By increasing the priorities of requests that failed the first time, we observe that prioritized requests never fail again. As a result of prioritization, the response times of requests with lower priorities increases. For example, compared to the red/right curve, which is for the implementation without prioritized queues (FIFO queues), a larger proportion of requests have response times between $40ms$ and $50ms$. In addition, requests with response times in the same cluster (e.g., centered around $45ms$) could have larger numbers of intermediate nodes (hops) in the prioritized vioson since prioritized requests have smaller response times per sub-request job.

V. Related work

Distributed hash tables are well known for their performance and scalability for looking up keys on a node that stores associated values in a distributed environment. Many existing DHTs use the number of nodes involved in one lookup request as a metric to measure performance. For example, Chord and Pastry require $O(logN)$ node traversals using $O(logN)$ routing tables to locate the target node for some data in a network of N nodes [15], [19], while D1HT [14] and ZHT [10] requires a single node traversal at the expense of $O(N)$ routing tables. However, this level of performance analysis is not suited for real-time applications, as these applications require detailed timing information to guarantee time bounds on lookup requests. In our research, we focus on analyzing response times of requests by modeling the pattern of job executions on the nodes.

We build our real-time DHT on the basis of Chord. Each node maintains a small fraction of information of other nodes in its soft routing table. Compared with D1HT and ZHT, where each node maintains the information of all other nodes, Chord requires less communication overhead to maintain its routing table when nodes or links have failed in the DHT. Thus, we believe a Chord-like DHT is more suitable for wide-area power grid real-time state monitoring as it is more scalable in the numbers of PDCs and PMUs.

In our timing analysis model, we assume that node-to-node network latency is bounded. Much of prior research changes the OSI model to smooth packet streams so as to guarantee time bounds on communication between nodes in switched Ethernet [9], [6]. Software-defined networks (SDN), which allow administration to control traffic flows, are also suited to control the network delay in a private network [12]. Distributed power grid control nodes are spread in a wide area, but use a proprietary Internet backbone to communicate with each other. Thus, it is feasible to employ SDN technologies to guarantee network delay bounds in a power grid environment [3].

VI. Conclusion

We have designed and implemented a real-time distributed hash table (DHT) on the basis of Chord to support a deadline-driven lookup service at upper layer control systems of the North American power grid. Our real-time DHT employs a cyclic executive to schedule periodic and aperiodic lookup jobs. Furthermore, we formalize the pattern of a lookup workload on our DHT according to the needs of power grid monitoring and control systems, and use queuing theory to analyze the stochastic bounds of response times for requests under that workload. We also derive the QoS model to measure the probability that the deadlines of requests can be met by our real-time DHT. Our problem was motivated by the power grid system but the cyclic executive and the approach of timing analysis generically applies to distributed storage systems when requests follow our pattern. Our evaluation shows that our model is suited to provide an upper bound on response times and that a prioritized extension can increase the probability of meeting deadlines for subsequent requests.

Our current analysis does not take node failures into consideration. With data replication (Chord stores data not only at the target node but in the nodes on the successor list [15]), requested data can still be obtained with high probability even if node failures occur. Failures can be detected when jobs on one node fail to send messages to other nodes. In this case, the job searches the finger table again to determine the next node to send a data request message to, which increases the execution times of that job. In future work, we will generalize executions time modeling so that node failures are tolerated.

References

[1] S. Bak, D.K. Chivukula, O. Adekunle, Mu Sun, M. Caccamo, and Lui Sha. The system-level simplex architecture for improved real-time embedded system safety. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 99–107, 2009.

[2] T.P. Baker and Alan Shaw. The cyclic executive model and Ada. Technical Report 88-04-07, University of Washington, Department of Computer Science, Seattle, Washington, 1988.

[3] A. Cahn, J. Hoyos, M. Hulse, and E. Keller. Software-defined energy communication networks: From substation automation to future smart grids. In *IEEE Conf. on Smart Grid Communications*, October 2013.

[4] T.L. Crenshaw, E. Gunter, C.L. Robinson, Lui Sha, and P.R. Kumar. The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures. In *IEEE Real-Time Systems Symposium*, pages 400–412, 2007.

[5] J. E. Dagle. Data management issues associated with the august 14, 2003 blackout investigation. In *IEEE PES General Meeting - Panel on Major Grid Blackouts of 2003 in North Akerica and Europe*, 2004.

[6] J. D. Decotignie. Ethernet-based real-time and industrial communications. *Proceedings of the IEEE*, 93(6):1102–1117, 2005.

[7] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[8] L. Kleinrock. *Queueing systems. volume 1: Theory*. 1975.

[9] S. K. Kweon and K. G. Shin. Achieving real-time communication over ethernet with adaptive traffic smoothing. In *in Proceedings of RTAS 2000*, pages 90–100, 2000.

[10] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[11] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[12] N. McKeown. Software-defined networking. INFOCOM keynote talk, 2009.

[13] W. A. Mittelstadt, P. E. Krause, P. N. Overholt, D. J. Sobajic, J. F. Hauer, R. E. Wilson, and D. T. Rizy. The doe wide area measurement system (wams) project demonstration of dynamic information technology for the future power system. In *EPRI Conference on the Future of Power Delivery*, 1996.

[14] L. Monnerat and C. Amorim. Peer-to-peer single hop distributed hash tables. In *in GLOBECOM'09*, pages 1–8, 2009.

[15] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[16] M. Parashar and J. Mo. Real time dynamics monitoring system: Phasor applications for the control room. In *42nd Hawaii International Conference on System Sciences*, pages 1–11, 2009.

[17] A. G. Phadke, J. S. Thorp, and M. G. Adamiak. New measurement techniques for tracking voltage phasors, local system frequency, and rate of change of frequency. *IEEE Transactions on Power Apparatus and Systems*, 102:1025–1038, 1983.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *in Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[19] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.