7 APPENIDICES

Appendix A: Overview of COACS Operations

In COACS, some nodes cache previously requested items, while another set of nodes index the queries along with the addresses of the caching nodes. As a result, there are two types of nodes in the context of COACS, caching nodes and query directories, although any of these two nodes can be a requesting node (RN, node that requests data items). Caching nodes (CNs) are nodes that store the cached items, and Query Directories (QDs) cache the queries along with the addresses of CN that host the answers of the queries. QDs are elected based on an algorithm that incorporates computing a score that incorporates the capabilities of the node, basically, the uptime, memory, battery, and bandwidth. At any time, where a QD goes offline, or the QD's score falls below a threshold, it gets reassigned.

In order for a node to request an item it scans QDs sequentially from nearest to farthest rather than the whole set of CNs, as the number of QDs in a network is much less than the number of CNs. When a query is found on a QD, the QD forwards the request to the CN that stores the item corresponding to the query. If all QDs are traversed, and the item was not found, there will be a cache miss. At this point, the last QD will request the data item from the server, and send it to the requesting node. The requesting node changes role to be a caching node (if it had not already been), while the nearest QD stores the query and the address of the CN.

It is worth noting, that in order to save on the traffic, the QDs store with each query its hash, and the CN indexes the items by the query hash, so that an RN requests an item using the hash rather than the actual request. In case there was a miss, the QD can then request the item from the data source as was suggested before. COACS acknowledges that QDs and CN have memory limitations and uses least recently used (LRU) as a cache replacement mechanism, in case no new CNs or QDs can be added. Moreover, COACS assumes a completely cooperative environment, where QDs willingly index queries as much as their memory allows. However, to motivate cooperation in the MANET, one can integrate a scheme that provides incentives for nodes to participate in a caching architecture, as in the case of [48], where a broker-based architecture was employed to provide incentives to nodes for replication and caching.

Appendix B: TTL adaptation

We describe the update or request process as a discrete random process: $U[n]=U[n\text{-}1]+IUI$, where IUI represents the inter-update (request) interval that is drawn from the exponential random variables described above, and $U[n]$ is the time of the n^{th} update at the server. After several polls and averaging the means of the estimated inter-update (request) Intervals using running averages, the TTL value will have the same mean as the inverse of the update (request) rate. This can be proved by considering the running average: $IUI=(1\text{-}\alpha)\times IUI+\alpha\times IUI_{meas}$, such that α is between 0.1 and 0.2, and IUI_{meas} has the same mean over steady periods, and assuming the initial value of IUI is IUI_{ini}. For simplicity we let $\beta=1\text{-}\alpha$, such that the above equation translates to $IUI=\beta\times IUI+(1\text{-}\beta)\times IUI_{meas}$. After introducing the mean operator on both sides we have $E(IUI)=\beta\times E(IUI)+(1\text{-}\beta)\times E(IUI_{meas})$. Since the measured $IUIs$ have the same mean, we get $E(IUI)=\beta\times E(IUI)+(1\text{-}\beta)\times 1/\lambda_u$. After n runs of this equation we have $E(IUI)=\beta^n\times IUI_{ini}+(1\text{-}\beta^n)\times 1/\lambda_u$. In fact, by setting $\beta=1\text{-}\alpha=0.875$, and after 10 iterations, the estimated mean of the inter-update interval will approach that of the update process on the server, and the effect of the initial value will be diminished. Similarly, we get the same result when estimating the request rate for the data items.

Appendix C: Time Response Gain

The analyzed response time is the average delay per data request. In the poll every time scheme, the data item is validated from the server each time it is requested. Then the response time estimated between the request and the answer is $T_{RTT}=T_{out}+T_{in}(2H_C+H_D)$. However in DCIM, the requested data item may have expired. If we assume that DCIM runs in full hit ratio mode, then all items are prefetched if necessary from the server, and thus all requests incur the same delay, except those that arrive while the item is being updated. This latter scenario represents the worst case as it incurs the same delay as the poll every time scheme. However, if the item is not expired, the query's answer is returned from within the MANET, where the response time is $T_{MAN}=T_{in}(2H_R+H_D)$. The latter delay is considerably smaller than the former one, but it is much more probable as will be shown later. To compute the average response time, we need to get the probability of each scenario. In fact, the first scenario of DCIM, i.e., the expired items scenario, occurs when the item is requested while it is being validated. This is equivalent to one request occurring in the update interval which was estimated above as $T_{RTT}=T_{out}+T_{in}(2H_C+H_D)$, and no updates in the last TTL interval (i.e., the request occurred while the item is being validated after its TTL expired). For a Poisson process, the above probability is simply $P_{SC1}=\lambda_U\times T_{RTT}\times e^{-\lambda_U\times T_{RTT}}\times e^{-\lambda_U\times TTL}$. However since we showed that on average $TTL=1/\lambda_U$, then $P_{SC1}=\lambda_U\times T_{RTT}\times e^{-\lambda_U\times T_{RTT}-1}$. It follows that the response time gain is: $GT=T_{RTT}-P_{SC1}\times T_{RTT}-(1-P_{SC1})\times T_{MAN}$.

Appendix D: Bandwidth Gain

We start by analyzing the poll every time scheme. If the item has changed on the server, the item must be fetched from the server, and the traffic is $B_{po}=S_R(H_D+H_C)+S_DH_C+S_RH_R$, where S_D is the size of the data packet. If the item is still valid, it is not fetched and the traffic is $B_{pi}=S_R(H_D+2H_C)$, where S_R is the size of the request. The probability in this scenario is equivalent to that of no updates during the request interval, which is $P_{poll}=e^{-\lambda_U\times T_R}$. The average traffic per request is then $P_{poll}\times B_{pi}+(1\text{-}P_{poll})\times B_{po}$. In the piggybacking interval T_{pigg} the probability of a single item getting requested k times is $P_R(k)=(1/k!)\left(\lambda_R\times T_{pigg}\right)^k e^{-\lambda_R\times T_{pigg}}$, then the average number of requests per single item is $\sum_{k=0}^{\infty}k\times P_R(k)=\lambda_R\times T_{pigg}$. For N items, each with re-

quest rate λ_{Ri}, the total number of requests is $R_{tot} = \sum_{i=1}^{N} \left(\lambda_{Ri} \times T_{pigg} \right)$. Then, the total bandwidth consumption by poll every time in a single piggybacking interval is $B_{poll} = R_{tot} \times (P_{poll} \times B_{pi} + (1 - P_{poll}) \times B_{po})$.

For DCIM, we present the average traffic in the piggybacking interval, then normalize the results for the poll every time scheme to compare to. We assume that this interval encompasses M polling intervals such that $T_{pigg} = M \times T_{poll}$, and that there are N total data items in the network. At the end of the polling interval, K out of N items are expired and must be validated at the server. Validating all expired items regardless of the request rate is merely a simplification for the analysis, and as mentioned before, this corresponds to the worst case scenario of DCIM. At the server, l items might have changed and the rest did not. The first part of the traffic is for the changed items and is denoted by $B_{pollc} = l \times (S_D H_C + S_R H_R)$, while the second is for items that did not change $B_{pollnc} = (K - l) \times S_R H_C$, where K and l will be determined later throughout this analysis, in addition to the request traffic: $B_{Rpoll} = K \times S_R H_C$. At the end of the piggyback interval all items need to be validated if at least one is expired. In such a case, m items might have changed on the server, and need to be prefetched at a total traffic of $B_{piggc} = m \times S_D H_C$. Also, the traffic induced from non-updated items is $B_{piggnc} = (N - m) \times S_R H_C$, where the request traffic is: $B_{Rpigg} = N \times S_R H_C$. In a polling interval, the average number of items to be validated depends on the number of items whose TTL expired in that interval. This is equivalent to items that did not get updated during the last TTL interval, which averages to $1/\lambda_u$ minus the duration of the last poll interval. The probability of a single item having its TTL expire in the last poll interval is $p_e = \int_{TTL - T_{poll}}^{TTL} e^{-\lambda_U \times t} dt = e^{-\lambda_U \times \left(1/\lambda_U - T_{poll}\right)} - e^{-1}$, where all items are assumed to have the same update rate for simplicity. Then, the average number of items to validate in a single polling interval is $K = \sum_{i=0}^{N} i \times \binom{N}{i} \times p_e^i \times (1 - p_e)^{N-i} = N \times p_e$. At the server side, out of the K items, there are l items that might have changed. An expired item changes at the server if it is updated in the last polling interval at least once, which happens with a probability of $p_c = 1 + e^{-1} - e^{-\lambda_U \times \left(1/\lambda_U - T_{poll}\right)}$. Hence, the average number of the changed items is $l = \sum_{i=0}^{K} i \times \binom{K}{i} \times p_c^i \times (1 - p_c)^{K-i} = K \times p_c$. However, at the piggybacking interval, all the items have to be validated, and similar to above, the average number of items to be updated is $m = \sum_{i=0}^{N} i \times \binom{N}{i} \times p_g^i \times (1 - p_g)^{N-i} = N \times p_g$, $p_g = 1 + e^{-1} - e^{-\lambda_U \times \left(1/\lambda_U - T_{pigg}\right)}$. As a result the bandwidth gain is stated as $GB = R_{tot} \times (P_{poll} \times B_{pi} + (1 - P_{poll}) \times B_{po}) - M \times (B_{Rpoll} + B_{pollnc} + B_{pollc}) - B_{Rpigg} - B_{piggnc} - B_{piggc}$.