An Evaluation of Consensus Latency in Partitioning Networks

Jason A. Tran*, Gowri S. Ramachandran*, Claudiu B. Danilov†, and Bhaskar Krishnamachari*
*USC Viterbi School of Engineering, University of Southern California, Los Angeles, USA
{jasontra, gsramach, bkrishna}@usc.edu
†Boeing Research & Technology, Huntington Beach, CA
{Claudiu.B.Danilov}@boeing.com

***Abstract*—Consensus, or state machine replication, is critical for the deployment of distributed battlefield systems. Battlefield networks operate in environments with unpredictable wireless connectivity which lead to sparse networks and frequent partitioning, and this makes deploying centralized architectures where nodes require a connection to a remote server unsuitable. The Extended Virtual Synchrony (EVS) model provides membership views which enables a network to reach consensus even after experiencing a series of partitions and mergers. If a node wants to propose state transitions that require nodes that are not currently in its membership view, then the node needs to wait until it reconnects with those nodes. The time the node has to wait to reconnect to the other nodes introduces consensus delays in the network. In this work, we evaluate consensus latency by focusing on these queued state transition proposals due to both network partition characteristics and distributed application/mission design. The key findings of our results show that consensus delay is least affected by network partitioning when the network splits at a rate equal to or less than $1/4$ the rate in which partitions merge. Our evaluation results provide application and mission designers guidelines on the tradeoffs between several network characteristics and desired consensus latency properties.**

***Index Terms*—Consensus, Ad-Hoc Network, Latency, Peer-to-Peer Network, Partitioning**

I. Introduction

Over the past few decades, an extensive collection of military applications have been developed and tested predominantly using the client-server architecture, wherein each device in the battlefield coordinates with a centralized server to perform its actions. Such an architectural pattern eases management complexity for application developers ensuring high availability and predictability in mission-critical applications.

In centralized applications, network connectivity to a server is critical. A server may crash or even be compromised by a malicious entity. Such issues lead to a single point-of-failure which is highly undesirable for military applications. If a group of nodes disconnect from the central server during a mission, the fraction of nodes should be able to continue coordinating and sharing information with one another. Decentralized architectures mitigate the need for a central server while paving the way for a self-managing application infrastructure for mission-critical applications. A fundamental building block of a decentralized application is a distributed consensus protocol, which allows the nodes in the network to collectively agree on a state of the system before performing any action.

For a distributed consensus protocol to agree on a system state, each node in the network has to participate in the consensus process. In typical distributed consensus algorithms, a sufficient number of nodes has to agree on a perceived system state to collectively make a decision. When achieving consensus in a battlefield scenario, nodes may leave or join the network due to the unpredictable nature of the wireless environment or mission itself. The intermittent wireless connectivity threatens the stability of the consensus process and may lead to different sets of nodes orphaned in inconsistent states. Achieving consensus in an intermittent network is, therefore, a challenging problem in decentralized applications. Battlefield environments rely on tactical networks which often do not provide ubiquitous connection to a cloud infrastructure like commercial distributed systems. Typical consensus protocols used in commercial, civilian deployments such as Paxos [1] and Raft [2] would fail if deployed in the battlefield. More specifically, these consensus protocols require a majority of nodes, or a quorom, to be connected in order to reach consensus. This would cause non-majority network partitions to block. Therefore, the underlying distributed consensus protocol in a battlefield network must support *dynamic membership views*, an abstraction where processes (or nodes) organize themselves into multicast groups to facilitate communication, in order to handle network partitions as they arise. Within a membership view, a reliable multicast service can be used with an ordered delivery protocol built on top to provide the foundation for reaching consensus on a system state.

The future battlefield is expected to be populated by much more autonomous systems including teams of autonomous ground and air vehicles. Certain mission objectives may require a specific subset of nodes with certain characteristics such as UAVs with particular capabilities or units with sufficient resources to carry out complex tasks like collaborative reconnaissance. Also, once these collaborative tasks are dispatched, the subset of pertinent nodes may encounter various network splits and mergers while accomplishing task objectives due to harsh communication environments and/or employing a divide and conquer strategy. While dynamic group memberships can offer flexibility, adopting this feature

This work was supported by Boeing Research and Technology.

inherently presents challenges to developing a distributed application which can guarantee reliable consensus in situations like these. To mitigate these kinds of issues, the first step to guaranteeing reliable consensus is to prevent conflicting state transitions in the face of network partitions. One solution is to adopt a protocol where a node that wants to propose a state transition be aware of the current partition membership. If the state transition requires specific nodes not currently present in said node's partition, the node can queue the proposal until the required nodes join its membership view. Adopting this partition aware protocol will inevitably introduce consensus delays in the network when nodes wait to reconnect with nodes outside of their partition to propose state transitions. To introduce stability in consensus delays, a mission can be designed such that teams attempt to rendezvous and reconnect by a target time deadline when the teams purposely split up or the communication environment causes the network to partition.

Key Findings: In this work, we investigate how expected network partitioning behaviors, namely the rate in which the network splits and merges, affect the latency of state transitions in a system. From our evaluation results, we observe that consensus latency is least affected by partitions when a network exhibits split rates less than or equal to $1/4$ the rate of mergers. Also, our results show that when nodes generate transactions (Tx) at a rate of $1Tx/s$ and the application design behavior generates transactions with a $\leq$ 10% probability that they are queued due to partition, the network size does not affect the average delay of transactions due to partitions. Our characterization results in this work provide application developers and mission designers guidelines on the tradeoffs of network characteristics and desired consensus latencies.

II. Background and Evaluation Focus

A. Consensus in the Battlefield

With the increasing demand of decentralized deployments in tactical applications, autonomous coordination and distributed decision making are key features for deploying future battlefield systems. Distributed consensus algorithms are critical for achieving deterministic outcomes within a group without a central coordinator. For example, a number of autonomous UAVs equipped with different sensors may need to decide on allocating roles and coordinating flight paths to ensure multiple targets of interest are tracked. Another example includes a mission objective which may separate a set of autonomous platforms into multiple teams which can carry out the mission even if the teams cannot communicate among one another. Achieving consensus is notoriously difficult in partitioning networks. This can intuitively be seen because reaching agreement requires knowledge of the needs of other nodes. Quantifying the time it takes to reach consensus, while needed for ensuring timeliness in mission development, is many times uncertain due to the unstable nature of connectivity among nodes in tactical deployments. However, missions can be designed to maintain certain bounds of intermittent connectivity. Thus, it is of interest to study consensus latencies in intermittent networks.

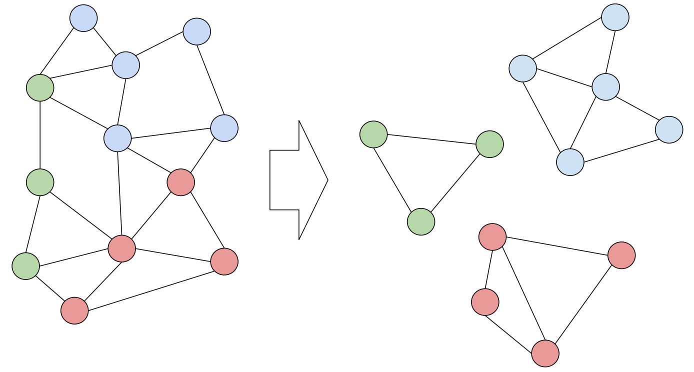

Fig. 1. Illustration of a network partitioning and creating several, independent group membership views.

B. Background

Tactical networks are sparsely connected and frequently encounter network partitions like illustrated in Fig. 1. *Extended Virtual Synchrony (EVS)* by Moser et. al [3] provides a model for managing *group membership views* where a set of processes (e.g., network nodes) can form groups and processes can join and leave these groups. EVS requires that messages sent by a process in a group be delivered reliably to all members within a group. That is, in any partition (such as the three partitions on the right side of Fig. 1), a message is only considered delivered if all processes in their respective groups receive that message. The EVS model also guarantees total ordering of messages within a group membership view, and preserves causality of messages across an entire network after experiencing network splits and mergers. When a networked system implements the components of the EVS model, a distributed application can successfully achieve consensus (i.e., state machine replication) even in the face of network splits and mergers.

Within a network partition, state machine replication (SMR) under the EVS model can be abstracted as follows. Nodes in the partition can reliably send ordered messages among one another to propose state updates. The state machine replicator uses the ordered messages to generate a log of *transactions (Tx)* and ensures that every non-faulty node sees the same log of transactions. A distributed application uses the log as an input to compute the current state of a system locally. In other words, nodes can propose transactions to be written to this log. This work focuses on transactions, not the messages required to create the log. The state machine replicator can be broken down into three basic building blocks as illustrated in Fig. 2: the group membership algorithm to manage partition views, a group multicast mechanism to send messages to all partition members, and a transaction ordering mechanism. This SMR abstraction will be used to simplify our system model.

A distributed application built on top of the EVS-based state machine replicator proposes transactions by sending messages

to all the members in the view using the group multicast mechanism. This is pictured in Fig. 2. When a transaction is finalized, it is passed to the distributed application in the same, ordered fashion at all nodes within a membership view. The group membership algorithm establishes the multicast groups and provides the distributed application updates on the membership view so the application is always aware of the current member list before it generates a transaction proposal.

Evaluation Focus: Nodes may want to propose new transactions which require specific nodes to be present. If they are currently not in the partition, the node will need to queue the transaction proposal *in a non-FIFO queue* (see "Tx Queue" object in Fig. 2) until it joins a membership view with all the required nodes. This work aims to evaluate how the characteristics of a partitioning network and the design of a distributed application affect the consensus latency of a system specifically due to these queued transactions. We accomplish this by measuring the time between a transaction entering the "Tx Queue" in Fig. 2 to the time it gets delivered to the distributed application running locally at each node.

Evaluation of consensus algorithms in the face of partitioning networks is not well studied in the literature. The authors of the BlockBench framework in [4] evaluate blockchain consensus algorithms and present throughput and latency results. Similarly, Androulaki *et al.* [5] evaluate the Hyperledger Fabric blockchain platform and present the consensus latency for various workloads. Unlike [4] and [5], our work evaluates consensus latency in the face of network partitions. Urban *et al.* [6] present a novel leader-based Paxos consensus algorithm and evaluate it in partitioning networks by crashing multiple processes. Our work is similar to [6], but we do not consider a particular consensus algorithm.

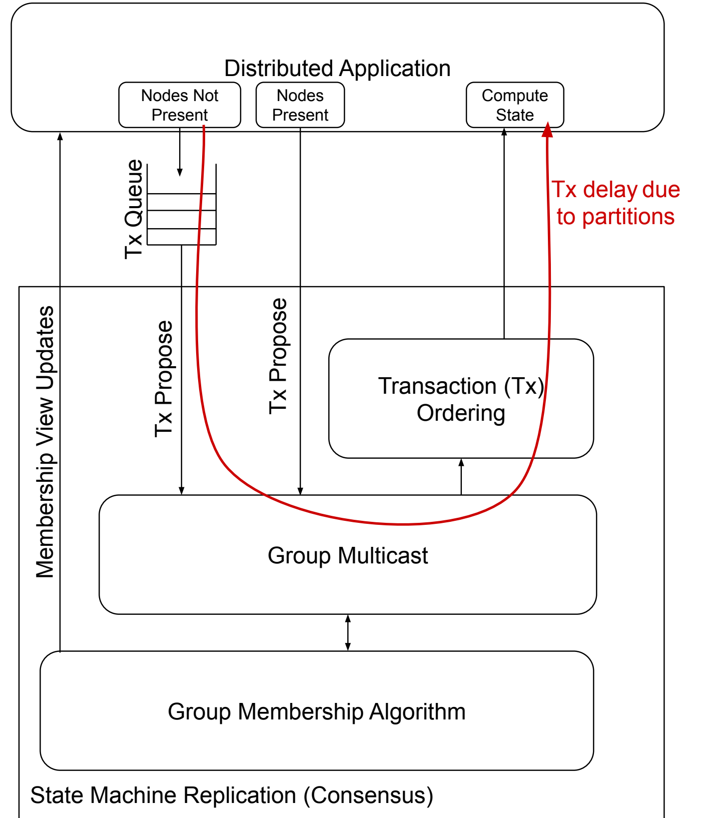

Fig. 2. Diagram of abstract building blocks of a distributed application built on top of Extended Virtual Synchrony (EVS) based state machine replication (SMR).

III. System Model

We consider a network of nodes that have prior knowledge of all members in the global set of nodes Π. All nodes run a group membership algorithm such as the one defined by Cachin et. al in [7]. At any node, the membership view $V = (id, M)$ is a tuple that contains a unique partition identifier id (such as a hash) and a set M consisting of the members of view id. Multiple views can exist at any instant of time (i.e., partitions). We assume all nodes in a membership view can communicate among themselves according to the EVS model via an arbitrary networking protocol as the networking protocol is not within the scope of this paper. Each message sent by a node is used to propose a new transaction. Any partition in the network at any time runs an instance of a state machine replicator and comes to consensus on a state by processing a log of transactions that are consistent across all nodes within a partition.

Transactions are proposed by any node with i.i.d. Poisson processes, and transactions arriving at any node in the entire network is modeled by a Poisson process with total rate λ_{Tx}. When a transaction arrives at any node, the probability of the event that the transaction requires a random subset of nodes not all present in the node's current partition is p_{queue}^{Tx}. That is, p_{queue}^{Tx} is the probability with which the transaction is placed into that node's pending transaction queue. Network splits (i.e., the splitting of a randomly selected partition) and network mergers (i.e., the merge of two randomly selected partitions) are modeled as Poisson processes with rate λ_{split} and λ_{merge}, respectively.

The partition status of the network can be modeled as an $M/M/1/N$ queue with an arrival rate of λ_{split}, service rate of λ_{merge}, and finite buffer size equal to the network size N. That is, if the buffer is completely full of network split event arrivals, then the network consists of N partitions. Under this model, the average percentage of time the network will be partitioned can be represented accordingly as $\rho = \lambda_{split}/\lambda_{merge}$. This means that the average lifetime of the network being partitioned, $E[T_p]$, can be modeled as the expected busy time of a $M/M/1/N$ queue, which is expressed as $E[T_p] = 1-(1-\rho)/(1-\rho^{N+1})$. As long as $\lambda_{split} < \lambda_{merge}$, the system will be stable. That is, the network will merge back into a single partition within a bounded time duration. Any battlefield network should be designed to always attempt to eventually reconnect, so we assume the network is always stable.

IV. Simulation Setup and Justification

To better study the characteristics of consensus delays, a network simulation tool written in Python is designed using the well known SimPy process-based discrete-event simulation framework [8]. Each node is not independently simulated, but rather the evolution of the network as a whole is simulated for reduced computational complexity. If each node is independently simulated, SimPy will create as many processes as there are nodes in the simulator causing unnecessary overhead. Instead, three processes are created, each generating events with exponentially distributed time intervals at different rates: a network transaction generator with rate λ_{Tx}, a network split generator with rate λ_{split}, and a network merge generator with rate λ_{merge}. Each simulation run is initialized with a partition table. When a network split or merge event arrives, the partition table is updated accordingly to reflect the current partition status of the network.

When a transaction event arrives, the simulator chooses a random node in the network to propose a transaction. All transactions that arrive have a probability p_{queue}^{Tx} with which the transaction requires a random subset of nodes with at least one node outside of the chosen node's current partition. This probability is set prior to simulation runs and stays constant throughout the run. The probability captures the behavior of the application logic and mission. An example of such a behavior is the local control logic at an autonomous node limiting the frequency in which the node attempts to dial nodes outside of its partition. Another example would be a mission design exhibiting a certain frequency of nodes attempting to dial other nodes outside of their partitions. If a transaction that arrives does require nodes outside of the current partition, the simulator will generate a random subset of nodes with at least one outside node and place the transaction into a queue along with a SimPy environment timestamp.

When a network merge event occurs, the simulator will pick two partitions at random and merge the two into a single partition. After the merge, the simulator will search through the transaction queues of all the nodes in the newly created partition and evaluate if any transaction proposals are valid. If valid, the transaction will be appended to the distributed log, a timestamp will be taken of the SimPy environment, and the delay of this transaction will be stored for analysis. If there is only a single partition in the network, then the merge event is ignored.

When any network split event arrives, the simulator will select a random partition in the network with at least two nodes and split that partition with each new partition containing a mutually exclusive random subset of nodes from the original partition. If all partitions are of size one (i.e., the number of partitions is equal to N, then the arrival of a split event is ignored (hence the queue model with a finite buffer).

In this system, the average delay of queued transactions will be lower bounded by $E[T_p]$, which is a function of the rate at which the network partitions λ_{split}. We study the consensus latency behaviors via simulations by particularly varying the λ_{split} rate of the network. For all simulation results in Section V, we fix the merge rate at $\lambda_{merge} = 1/60$ (equivalent to an average of one network merge per minute). The merge rate is chosen purely for illustrative purposes. The actual split and merge rates can vary depending on the application, and our simulation results in the next section shed light on the relationship of consensus latency with respect to the ratio between the two rates. More specifically, our results show certain bounds on partition-related consensus delays when the split to merge ratio can be estimated. Nonetheless, merge rates faster than $1/60$ would indicate that a network will encounter partitions with an average lifetime less than 60 seconds. At a merge rate this fast, it is best to deploy a consensus algorithm that will block in the face of partitions rather than trying to adapt to fast changing membership views which can cause large network overhead, especially in wireless settings.

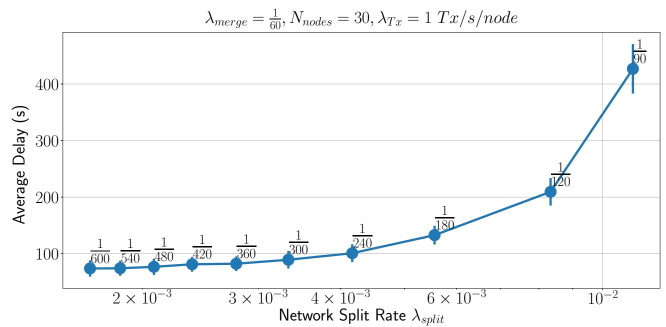

Fig. 3. Average delay of transactions that were queued due to a network partition as a function of the network split rate. Point labels indicate the corresponding λ_{split} and the horizontal axis is log scaled.

Generally speaking, network delays tend to increase with network size. However, simulating consensus latencies that result from communication delays is difficult and often does not accurately reflect networking delays of a real system. In addition, the actual latency of consensus relies heavily on the consensus algorithm utilized such as the two-phase commit algorithm [9] or practical Byzantine fault-tolerant algorithm (pBFT) [10]. Therefore, we do not include the network delays in our simulations, and we focus on consensus latencies introduced by queued transactions due to a network undergoing a series of splits and mergers.

V. Simulation Results

In all figures in this section, a single point is generated by simulating across a total time of 86400 seconds, or 24 hours. The bars in all of the simulation result graphs in this section represent the standard error.

Figure 3: The first simulation results is a graph of the average delay of transactions that were queued due to a network partition as a function of the network split rate. To be more precise, the average delay here is calculated strictly using the delays experienced by transactions queued at a node. It does not average across the total number of transactions that arrived in a simulation run. In this plot, it can be seen that as the split rate comes closer to the system's expected merge rate, queued

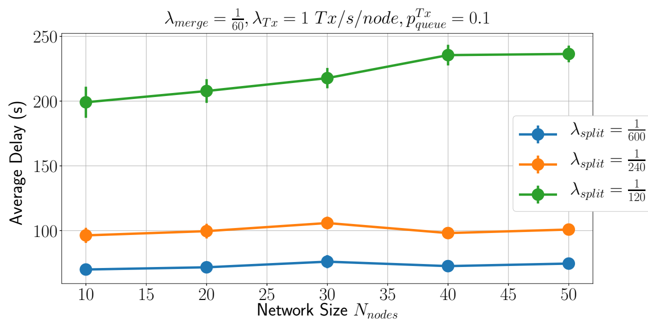

Fig. 4. Average delay of transactions that were queued due to a network partition versus network size for several network split rates.

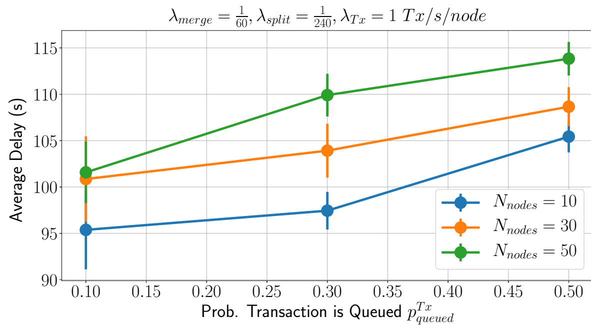

Fig. 5. Average delay of transactions that were queued due to a network partition with respect to the probability that a node generates a transaction that requires nodes outside of its current partition.

transactions will incur exponentially increasing average delays. This result motivates, as a rule of thumb, distributed battlefield applications should be constructed so that network partitions occur at a rate of at most $1/4$ of the network merge rate in order to optimize for low, stable delays. However, if a mission expects longer durations of network partitions, then an application can expect exponentially increasing consensus latencies. The behavior of this exponential growth is captured in the figure.

Figure 4: In this set of simulations, we investigate the effects of the network size on the average delay of queued transactions. We fix p^{Tx}_{queue} to 0.1 and run simulations with different network sizes and split rates. The results show that at split rates equal to $\frac{1}{4}\lambda_{merge}$ or less, the average delay of transactions is unaffected by network size. It can be seen that the average delay is higher with larger split rates, which is as expected after the results from Figure 3. However, at sufficiently high split rates, the average delay of transactions starts growing with network size. Our simulation results for this graph shows that battlefield applications, such as those involving autonomous UAV teams, should be designed to have split rates of at most $\frac{1}{4}\lambda_{merge}$ to minimize additional queued transaction delays due to network size. That is, large networks should be designed to not split the network (e.g., divide and conquer tactics) too frequently to prevent the network size from adversely affecting consensus delays.

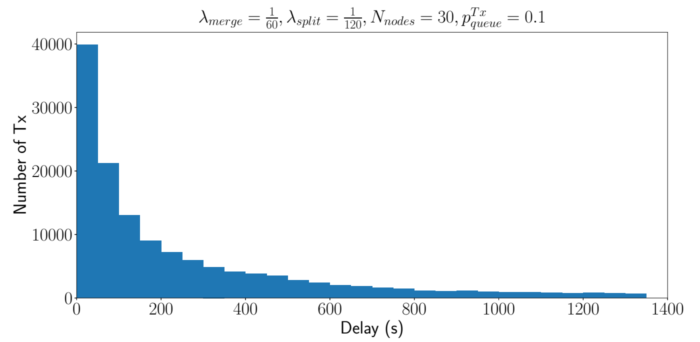

Fig. 6. Histogram of the delays of queued transactions for $\lambda_{split} = \frac{1}{120}$.

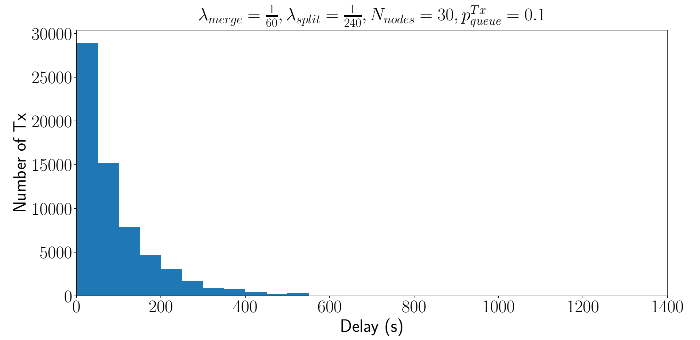

Fig. 7. Histogram of the delays of queued transactions for $\lambda_{split} = \frac{1}{240}$.

Figure 5: The local application logic at a node and the mission objectives may dictate the frequency in which a single node will create transactions that require nodes outside of the current local partition. To capture these system effects, we investigate the effect of this frequency on the average queued transaction delays by simulating several p^{Tx}_{queue} values. We discovered that for sufficiently large split rates such as $\lambda_{split} \geq \frac{1}{4}\lambda_{merge}$, larger p^{Tx}_{queue} values actually increase the average delay. This can be seen by the divergence of the average delays as p^{Tx}_{queue} increases. At $p^{Tx}_{queue} = 0.1$, we saw the average delay unaffected by the network size (see Figure 4). However, the results in these simulation runs show

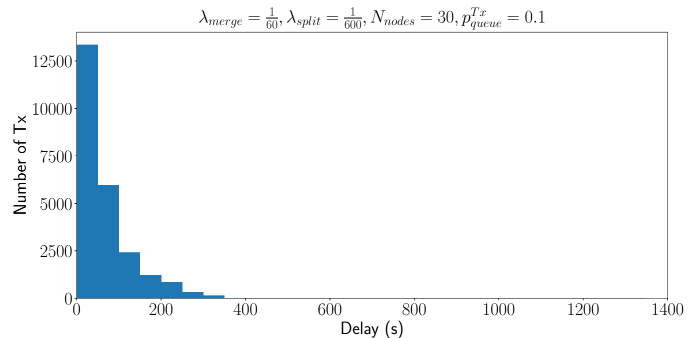

Fig. 8. Histogram of the delays of queued transactions for $\lambda_{split} = \frac{1}{600}$.

that network size begins to affect consensus latency when the frequency of queued transactions is increased. This effect persists even with $\lambda_{split} = \frac{1}{4}\lambda_{merge}$, which is the suggested threshold of the split rate for minimizing consensus latency. To be safe, if the p_{queue}^{Tx} of the autonomous system is larger than 0.1, designers should prioritize lowering the system's expected λ_{split} beyond the suggested split rate threshold.

Figures 6,7, and 8: A distributed application may want to support large delays introduced by partitions. We thus study the distribution of delays of queued transaction at various split rates, $\lambda_{split} = [\frac{1}{120}, \frac{1}{240}, \frac{1}{600}]$, by plotting the histogram of all delays in a single simulation run. While the average of queued transactions for $\lambda_{split} = \frac{1}{120}$ is 209.4 seconds, Fig. 6 shows that the tail of the distribution stretches as far as transactions with delays of up to $\sim$ 1400 seconds, or $\sim$ 23 minutes, which may be undesirable for short mission durations. The split rates of $\frac{1}{240}$ and $\frac{1}{600}$ show similar distributions of delays further motivating the design rule of maintaining the relationship of $\lambda_{split} \leq \frac{1}{4}\lambda_{merge}$.

VI. Conclusion

There is a strong demand for the deployment of more autonomous, decentralized systems in the battlefield. Designing consensus algorithms for tactical networks which are expected to experience frequent partitioning events is undoubtedly challenging. This work provides a state machine replication abstraction which allows for the evaluation of per-transaction consensus delays as a result of nodes waiting to reconnect with other nodes. The evaluation results in this work provide mission developers a better understanding of the tradeoff characteristics of network partition rates and transactional consensus delays. While our results show that systems which expect split rates of $\leq \frac{1}{4}\lambda_{merge}$ are least affected by partitioning events and network size, mission designers can plan more network splits if the objectives can tolerate the exponentially increasing transaction delays and variance of transaction delays.

References

[1] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[2] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666

[3] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, "Extended virtual synchrony," in *14th International Conference on Distributed Computing Systems*, June 1994, pp. 56–65.

[4] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1085–1100.

[5] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538

[6] P. Urban, N. Hayashibara, A. Schiper, and T. Katayama, "Performance comparison of a rotating coordinator and a leader based consensus algorithm," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, Oct 2004, pp. 4–17.

[7] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

[8] SimPy. [Online]. Available: https://pypi.org/project/simpy/

[9] G. Samaras, K. Britton, A. Citron, and C. Mohan, "Two-phase commit optimizations and tradeoffs in the commercial environment," in *Proceedings of IEEE 9th International Conference on Data Engineering*, April 1993, pp. 520–529.

[10] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.