# Tactical Jupiter: Dynamic Scheduling of Dispersed Computations in Tactical MANETs*

A. Poylisher, A. Cichocki, K. Guo, J. Hunziker, L. Kant
*Peraton Labs*
150 Mount Airy Road, Basking Ridge, NJ 07920

B. Krishnamachari, S. Avestimehr, M. Annavaram
*University of Southern California*
3650 McClintock Ave, Los Angeles, CA 90089

***Abstract*—We present Tactical Jupiter, an adaptation of the recently developed Jupiter framework for scheduling of dispersed computations on heterogeneous resources to tactical MANETs. Tactical Jupiter addresses the challenges to distributed scheduling posed by intermittent connectivity and scarce/variable bandwidth, variable computational resource utilization by background load, and node attrition. Our key contributions include: (a) disruption handling via increased autonomy of task executors, (b) low-overhead ML-based task completion time estimation in presence of background load, and (c) resilient dissemination mechanisms for monitoring information.**

***Index Terms*—Dispersed computing, scheduling, MANET.**

## I. Introduction

Recent advances in the computing power, storage, and energy storage capacity available at the tactical edge enable a significant distributed processing capability for the distributed data produced at the tactical edge. Harnessing the frequently unused part of this capability close to the data sources would greatly reduce the data transfer time and required network capacity and shorten computation time, leading to a higher quality and speed of decision-making.

Approaches for scheduling dispersed computations have been actively developed for the data center, fixed infrastructure IoT (e.g., [1], [2] and Internet-wide scenarios. However, at the tactical edge, they face multiple challenges, including (a) intermittent connectivity and low, variable bandwidth, due to mobility and/or adversarial action, (b) variable computational resource utilization by the legacy users not subject to distributed scheduling, (c) node attrition due to adversarial action.

In this paper, we present Tactical Jupiter, an adaptation of a recently developed USC Jupiter framework for scheduling of dispersed computations expressed as task graphs on heterogeneous resources [3], [4], to tactical MANETs to address a subset of above challenges. A task graph (TG) is a directed acyclic graph where a vertex is a computational task and an edge is a data transfer between tasks. The TG scheduling problem is to produce a mapping or a series of mappings of tasks in a TG to network nodes to minimize the total execution time of the TG. Tactical Jupiter uses a modified version of the HEFT algorithm [5], and continuous feedback from monitoring of the compute network resources and executing tasks and data transfers to regularly evaluate and adjust the TG mapping to nodes in a tactical MANET.

Our key contributions include: (a) handling of selected disruption types via increased autonomy of task executors, (b) low-overhead ML-based task completion time (TCT) estimation in presence of background load, and (c) resilient and low-overhead dissemination of monitoring information.

The rest of the paper is organized as follows. In Section II, we describe the system architecture, the dynamic task mapping, task completion time estimation and path available bandwidth estimation approaches. Section III describes a few implementation details and presents initial evaluation results for task completion time estimation and the overall system. Section IV discusses related work, and Section V concludes and outlines our next steps.

## II. The Tactical Jupiter Approach

### A. Architecture

The architecture of Tactical Jupiter is shown in Fig. 1. Tactical Jupiter extends Jupiter, the USC framework for scheduling and executing dispersed computing applications [3], [4], including those created with coded computing mechanisms [6], to address the challenges posed by the resource dynamics in tactical MANETs, outlined in Section I. Given an application described by a a task graph (TG), and a tactical network that connects computational nodes, Tactical Jupiter dynamically maps the tasks for execution on the nodes to minimize the overall TG execution time.

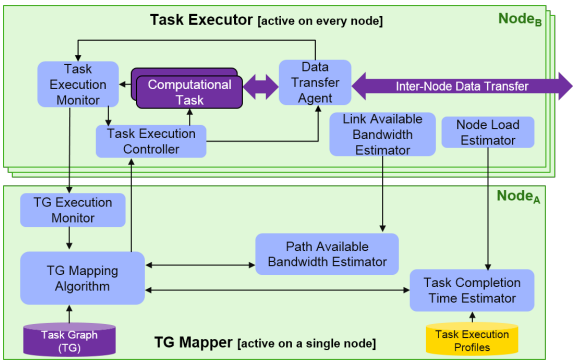


Fig. 1. The architecture of Tactical Jupiter. Runtime modules are in blue, the application TG in purple, and offline-computed reference data in yellow.

*This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-17-C-0047. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

Tactical Jupiter is composed of two high-level components: (a) the TG Mapper that is active on a single network node, and (b) the Task Executor that runs on every network node configured to execute computational tasks. The TG Mapper is responsible for: (a) computing new/updated assignments of tasks to nodes (mappings) based on various events, including arrivals of new TGs, changes in resource availability, and task completions/under-performance, and (b) pushing out new/updated mappings to the Task Executors on the nodes identified to execute the tasks. The TG Executor is responsible for: (a) instantiating/terminating tasks, (b) monitoring execution of ongoing tasks and informing the TG Mapper of task completion/under-performance.

The brain of Tactical Jupiter is the TG Mapping Algorithm that can implement any Jupiter algorithm (e.g., [5] or [7]); ours implements an modification of [5]. The TG Mapping Algorithm has the following inputs: (a) path available bandwidth for any node pair from the Path Available Bandwidth Estimator, (b) projected task completion time (TCT) for a task on any node from the TCT Estimator, and (c) mapped task progress information from the TG Execution Monitor.

The Path Available Bandwidth Estimator maintains an annotated network topology and is informed by Link Available Bandwidth Estimators that run on every Task Executor node. The TCT Estimator maintains node computational load information, informed by Node Load Estimators that run on every Task Executor node. The TCT Estimator also makes use of the pre-computed task execution profiles (Section II-C).

The TG Execution Monitor is informed by Task Execution Monitors that run on every Task Executor node. The TG Mapping Algorithm effects mapping changes via Task Execution Controllers that run on every Task Executor node. The Task Execution Controller starts computational tasks and inter-task data transfers autonomously, when these are ready, according to the current mapping. The Data Transfer Agent implements the application-independent inter-task data transmission and supplies progress information to the Task Execution Monitor.

In the rest of this Section, we describe the operation of Tactical Jupiter, including task mapping (Section II-B), task execution profiling and TCT estimation (Section II-C), and path available bandwidth estimation (Section II-D).

### *B. Task Mapping*

The TG Mapping Algorithm depends on the following inputs: (a) the TG, (b) the timeline of previously-scheduled task assignments for each executor node, (c) the expected TCT for each task on each executor node, (d) the expected size of data transferred by each task upon completion, (e) the estimated available bandwidth between all pairs of executor nodes, and (f) the state of in-progress data transfers.

Upon generation of the initial mapping, the timeline of availability for each executor node is empty. Note that the timeline does not track the computational load unrelated to the TGs managed by Tactical Jupiter. The TCT Estimator (Section II-C) handles the effects of such background load. The use of persistent executor node-oriented timelines allows multiple TGs to run at once. Each TG has its own mapping of tasks to executor nodes.

```
for node ← all new compute nodes do
    Set timeline to empty
end for
if first schedule then
    Record kickoff time
end if
Use HEFT (1) to calculate up-ranks of all tasks
for task ← tasks sorted by decreasing up-rank do
    for node ← all compute nodes do
        Use HEFT (2) and (3) to calculate earliest start time and earliest finish time,
        greedily assigning the node with the best earliest finish time to each task
    end for
end for
```

Fig. 2. Task mapping with HEFT.

To determine a mapping of tasks to processors, the basic HEFT algorithm operates as shown in Fig. 2 and 3.

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)) \quad (1)$$

$$EST(n_i, p_j) = \max \left\{ avail[j], \max_{n_m \in pred(n_i)} (AFT(n_m) + c_{m,i}) \right\} \quad (2)$$

$$EFT(n_i, p_j) = w_{i,j} + EST(n_i, p_j) \quad (3)$$

Fig. 3. HEFT equations: 1) Up-rank equation, where $n_i$ is the $i^{th}$ task, $\overline{w_i}$ is the average computation cost of $n_i$ across all nodes, $succ(n_i)$ is the set of successors to $n_i$ in the TG, and $\overline{c_{i,j}}$ is the average transfer cost of sending data between $n_i$ and $n_j$ for all possible node pairs. 2) Earliest Start Time equation, where $p_j$ is the $j^{th}$ processor, $avail[j]$ is the earliest slot in $p_j$'s timeline where this task would fit, $pred(n_i)$ is the set of predecessors to $n_i$ in the TG, and $AFT$ is the Actual Finish Time of such a predecessor. Note that "actual" here means the finish time implied by the schedule when the predecessor has not yet run. AFT is the EFT of the chosen node for a task. 3) Earliest Finish Time equation, where $w_{i,j}$ is the computation cost of running this task on this node.

Because of the changing TCT estimates and network resource availability, the TG Mapping Algorithm periodically recomputes the the current mapping. If the new mapping is a significant improvement over the existing one (in terms of estimated completion time and taking into account the cost of data transfers, including re-mapping), new task assignments are put into effect. In the tactical networks, both the nodes and the communication links can be unreliable. When the TG mapping changes, the challenge is to ensure that tasks communicate with the correct predecessors and successors. If the TG Mapper were responsible for canceling tasks and re-sending data, race conditions would become a significant risk. Particularly difficult situations could arise when the Task Execution Controllers miss important control messages from the TG Mapping Algorithm.

The key insight into handling disruptions such as node disconnections and nodes failing to expediently complete tasks is that *autonomy in the execution nodes relieves the scheduler of the most complex responsibilities*. A philosophy of "the task assignment map drives everything" allows for graceful adaptation to changes. The task assignment map is the output of HEFT, described above. In Tactical Jupiter, the full assignment map is sent to every executor node. By having the whole assignment map, a Task Execution Controller can do a number of things autonomously, including: (a) beginning

```
loop
    if all inputs have arrived for an assigned task then
        Put task on job queue
    end if
    if new task assignment map arrives then
        for task ← tasks that have been completed here do
            if successor has changed then
                Send old result data to new successor
            end if
        end for
        if one of this node's assignments have changed then
            Cancel execution of that task
            Cancel result transfers for that task
        end if
    end if
end loop
```

Fig. 4. Task execution. Note that there are separate threads handling data transfer and execution of tasks on the job queue.

```
loop
    for task ← all tasks do
        if processor has exceeded expected time for task then
            Mark task as eligible for rescheduling
        end if
    end for
    Run scheduling algorithm (Fig. 2) with updated network and computational load information
    Estimate previous schedule with updated network and computational load information
    if proposed schedule finishes significantly sooner than previous schedule then
        Adopt proposed schedule
    end if
    Send schedule, regardless of whether it has changed, to all nodes
end loop
```

Fig. 5. Periodic mapping re-computation.

task execution when the necessary inputs arrive, (b) sending previously computed outputs to any executor node to which results have not yet been sent, (c) interrupting task execution if the current assignment is revoked, and (d) interrupting a data transfer if the destination node's assignment is revoked. This set of behaviors is illustrated in Fig. 4.

With the "the task assignment map drives everything" approach, the TG Mapping Algorithm effects all assignment changes solely by transmitting a new assignment map. It does not get involved in starting and stopping tasks, or orchestrate network transfers. In the event of the disconnection of a large number of executor nodes, the system gracefully recovers.

Keeping currently executing tasks on their assigned nodes is a virtue for system stability and ease of understanding. The TG Mapping Algorithm does not normally re-map the tasks in progress. However, if a predicted TCT is exceeded by a configurable threshold, the task becomes eligible for re-mapping. This eligibility cascades up the TG if predecessor task outputs are no longer available.

Fig. 5 shows the procedure of periodic mapping re-computation. The current mapping is re-evaluated after the path available bandwidth and executor node load information has been collected. This is because improving conditions can benefit the current mapping, so the appropriate comparable for a new mapping is the re-evaluated current mapping. The resulting mapping is sent to all executor nodes regardless of whether it has changed, so that the Task Execution Controllers that miss mapping messages (e.g., due to network instability) get the correct schedule upon reconnecting. Disconnected nodes continue operate on an outdated schedule, and can make progress if data is available. Their output that can be used or discarded, but in no way affects correctness of the end result.

In summary, our TG Mapper improves upon the original Jupiter by: (a) re-mapping during TG execution, (b) handling node losses and task execution timeouts, (c) using dynamic TCT estimation for tasks in the currently executing TG, and (d) pipelining multiple TG executions.

### C. Task completion time (TCT) estimation

To make the best use of CPU resources on each compute node, a scheduler typically relies on prediction of future performance. Existing work on CPU availability prediction is often motivated by the need of the scheduler to assign a task to the node with highest projected CPU availability [8], [9], the percentage of CPU time available to a new process.

Our TCT Estimator takes this one step further and predicts the time it would take to complete the execution of a given profiled task were the task to execute on a particular node in the immediate future. This is challenging as task execution requires nonzero time, and, during this time, CPU availability varies as background CPU load changes. We make no assumption about the background load.

*Detecting the effect of background load change*: Consider the average behavior of a foreground process to be scheduled, and the set of background processes as aggregate background load. Further, consider the typical case where all processes share CPU resources of one or more cores/CPUs fairly. With $n$ concurrent processes competing for CPU resources, each will receive $1/n$ of the total CPU resources. We use $r_f$ and $r_b$ to represent the average number of competing CPU-bound processes during the time period under consideration for the foreground process and the set of background processes.

To obtain these metrics, we use the OS scheduler statistics provided by the in-kernel instrumentation. E.g., for Linux and Android, we sample the *procs_running* entry in */proc/stat* for the instantenous reading of the run queue size, which reflects the number of running processes, and average such readings over a time window for a smoothed average run queue size. To reduce overhead, we use a 1s sampling interval.

However, the CPU scheduling quantum (the period of time for which a process is allowed to run uninterrupted in a preemptive multitasking OS) is typically in the order of milliseconds (e.g., 0.75 to 6 ms for the Linux time slice). At time scales larger than the time slice, the run queue size alone does not fully represent the CPU load incurred by the group of processes under consideration. This drives the need to monitor *CPU utilization*. We obtain per-process CPU utilization from the */proc* filesystem as well, sample it every 1s, and average the per-second reading to derive a smoothed average CPU utilization for a given window. We define $u_f$ and $u_b$ as the average CPU utilization over a window for the foreground task and all of the background processes.

Consider a window when both a foreground task and background processes are running. Since under fair CPU sharing CPU load from a process is proportional to its average run

queue size, the foreground task should receive $r_f/(r_f+r_b)$ as its share. When the foreground task is not CPU intensive, $u_f$ is less than $100\%$, and its TCT is less affected by the background load. The minimum TCT is $TCT_0$, obtained without any background load. We define the *TCT scaling factor* for a foreground task as the ratio of TCT with background load over $TCT_0$. The minimum TCT scaling factor is 1, and the maximum TCT scaling factor is reached when the foreground task is CPU-intensive during the entire execution time with $u_f = 100\%$.

$$max(1, u_f \times (r_f + r_b)/r_f) \tag{4}$$

Next, we discuss *task execution profiling* and *background CPU load estimation* that provide inputs to the TCT Estimator, and the process of *TCT estimation*.

*Task execution profiling*: By design, foreground tasks are known to the scheduler in advance, and background processes are unknown. To profile a foreground task, we execute it multiple times in a row without any background load, on each machine representative of an executor node in the actual network, and record the TCT, average run queue size and average CPU utilization for each execution. We then average over all executions to derive $TCT_0$, $r_{f'}$ and $u_f$. Since our monitoring process reads a small number of files every 1s, it makes the contribution of 1 to the measured $r_{f'}$ and negligible contribution to CPU utilization. The profile run queue size for the foreground task is therefore $r_f = r_{f'} - 1$.

*Predicting background CPU load*: Due to the unknown nature of background processes, we can rely only on prediction to capture average background CPU load, using CPU utilization $u_b$ and run queue size $r_b$ over a time interval. We record the two metrics periodically (every 1s), and solve the time-series prediction problem using a bi-directional LSTM network [10] for each metric. We select the average value over the past time window $W$ as the input feature, and the average value over the next time window of the same size as the output. We train the model with traces from a number of benchmark tests from the Phoronix test suite [11], and test with traces not included in the training dataset. With $W \geq 120$s, we achieve 80-90% accuracy (or 10-20% prediction error). We use RMSE over all predictions for scaled input data in the range of [0,1] in order to derive the percentage error. For reference, the same error range has been reported in a recent large data center server trace study [12].

*TCT estimation*: Foreground TCT prediction using the TCT scaling factor in Eqn. 4 requires the foreground task profile, namely $u_f$, $r_f$, and $TCT_0$. For background processes, it only requires prediction for $r_b$ for the next time interval $W$. In order to predict $r_b$, the input feature to the prediction model must be the average $r_b$ in the previous interval. However, when both the foreground process and background processes are present, our monitoring process records the run queue size and CPU utilization for all processes, including the monitoring process, the foreground task, and the background processes.

During $W$, the monitoring process is always present. In contrast, the foreground task may only be present for a portion of $W$, or not present at all. Because the TG Mapping Algorithm records the start and end time of each scheduled task on any executor node, the TCT Estimator computes $p$, the percentage of $W$ the task is running.

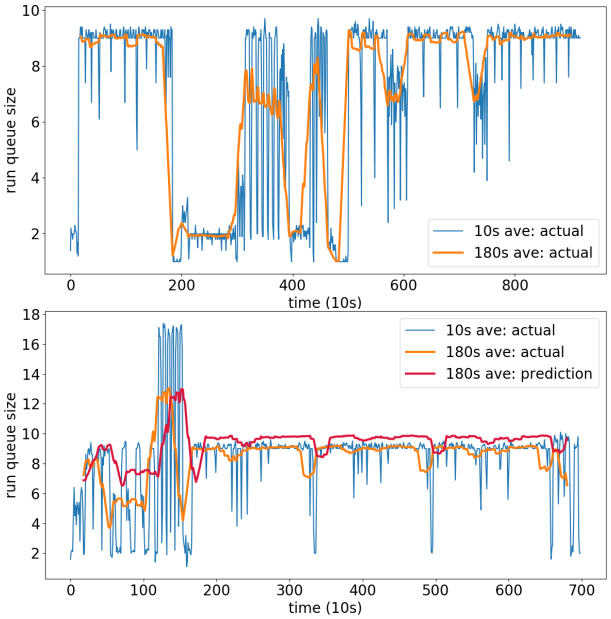


Fig. 6. Run queue size trace for training (top) and prediction results (bottom).

If $r_c$ is the combined run queue size for $W$, the input to the $r_b$ prediction model becomes:

$$r_b = r_c - 1 - p \times r_f \tag{5}$$

### D. Path bandwidth availability estimator

When generating a schedule, the scheduler considers available bandwidth between all node pairs. Estimating end-to-end available bandwidth in tactical MANETs presents a few challenges. First, unlike in the wired networks, links may not be independent due to spectrum interference, which is particularly significant for CSMA MANETs we have used to evaluate Tactical Jupiter. Second, combined effects of mobility and terrain require frequent re-estimation.

The available bandwidth of a CSMA MANET link is a function of: *a*) the traffic carried by the link in consideration; *b*) the traffic carried by the other links that can interfere on the same channel; *c*) potential spectrum interference from outside the waveform; *d*) the PHY and MAC layer construction and configuration (e.g., the transmission and carrier-sense ranges); and *e*) node positions and terrain.

We have developed a low-overhead sensing-based path bandwidth estimation approach for 802.11 MANETs that takes advantage of a well-known MANET link state routing protocol [13]. In our approach, described in detail in [14], each node A periodically: *i*) estimates the probability of its own 802.11 MAC being idle; *ii*) shares its MAC idle probability with one-hop neighbors by piggybacking on the periodic link state protocol HELLO messages; *iii*) estimates one-hop available link bandwidth between a neighbor and itself (L) as the product of the minimum MAC idle probability of the two nodes and maximum channel capacity; *iv*) efficiently disseminates

L throughout the network using the topology control (TC) messages of [13], so the topology annotated with L is available at each node; and *v*) adjusts the estimate at A to account for intra-flow contention and interference by dividing the estimate by the number of nodes within the carrier sensing range of A. *v*) requires the knowledge of node positions, terrain, and radio configuration, available in tactical MANETs.

The available bandwidth along a path is then computed from the annotated topology on demand at any node, including the node currently hosting the TG Mapper, as the minimum link bandwidth estimate along the path. This capability improves fault tolerance and facilitates task re-mapping; we have implemented it as a service in the routing protocol daemon of [13].

## III. Implementation and Experimental Evaluation

Tactical Jupiter is implemented in Python, runs inside Docker containers on Ubuntu and Android, supports any application expressed as a TG, and custom eligibility criteria for task placement. The latter is useful for nodes that can act as sensors or have custom processing capabilities.

We have evaluated Tactical Jupiter in a CyberVAN hybrid emulation testbed [15] that combines high-fidelity discrete event network simulation with virtual machines (VMs) running real operating systems, libraries and applications.

*TCT Estimator*: To test the performance of our TCT Estimator, we designed a number of foreground tasks with TCT ranging from 60 to 180s with average CPU utilization $u_f$ ranging from 10% to 100% on a 1-core and a 8-core x86_64 virtual machine (VM) running Ubuntu 16.04. We varied the task execution environment from native on a VM to inside a Docker container. We used the Phoronix test suite [11] to generate background load. Our TCT Estimator is able to achieve 87-95% accuracy in these tests. Figure 7 shows the results from profiling an example task over 100 iterations and the TCT estimation results for run queue size and TCT for a foreground task with background load.

*Full system evaluation*: We conducted initial evaluation of Tactical Jupiter in a CyberVAN scenario (Figure 8). The scenario includes a low-bandwidth, platoon-sized 802.11n MANET deployed on a relatively flat terrain. Node computational capabilities are heterogeneous (handheld devices and vehicle computers). Resource availability changes due to node mobility and background CPU load.

The application TG (in purple) represents a multi-stage continuous training of autoencoder-based models and building of ensemble classifiers to detect anomalies in the SA reports sent by the ATAK application on all nodes that can be used as a covert coordination channel by an adversary.

In our baseline scenario, an initial TG mapping is created by hand, but there is no run-time re-mapping. In contrast, in the Tactical Jupiter scenario, the TG Mapper performs dynamic adaptation in response to: (a) a node destroyed by enemy fire, and (b) a spike of background CPU load on a different node. In the baseline scenario, the overall TG completion time was severely affected by (b), but, even more significantly, the computation could not finish at all due to a later-stage task

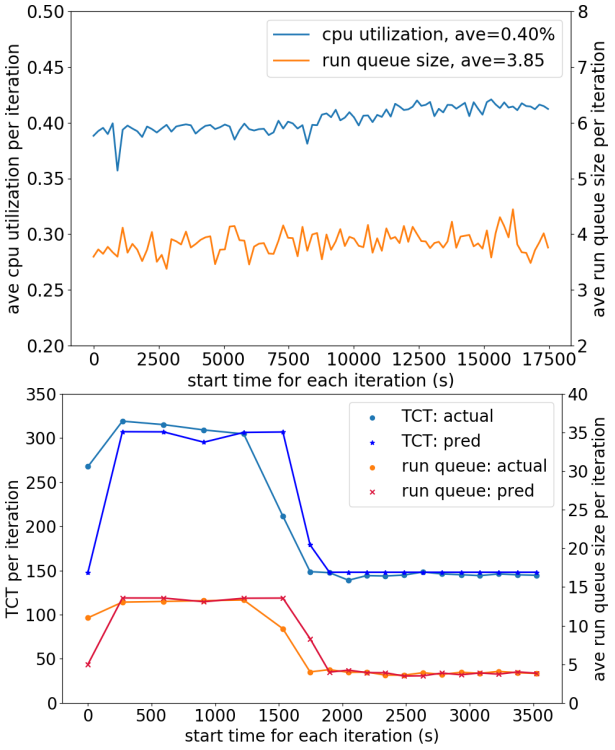


Fig. 7. TCT estimation results based on task profile metrics: (top) task profile: CPU utilization and run queue size averaged over 100 task iterations, (bottom) prediction for run queue size and TCT for 19 task iterations.

mapped to a destroyed node by the static mapping. Tactical Jupiter re-computed the mapping *within 10s* in both cases, and the TG completed, avoiding the large delay due to (b).

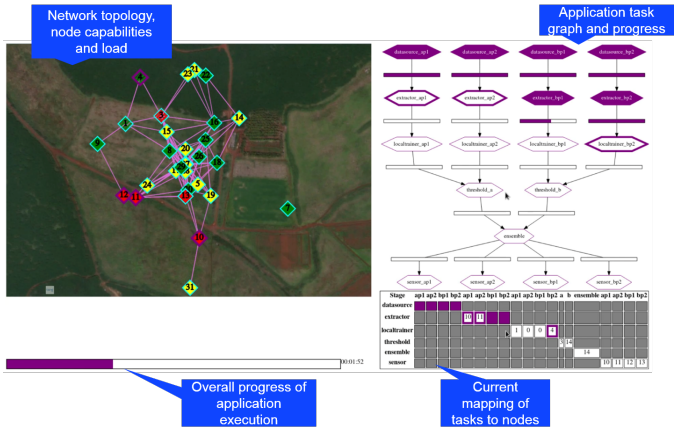


Fig. 8. The CyberVAN evaluation scenario.

## IV. Related Work

*Task mapping*: Prior work in the area of scheduling distributed computations have largely been based on heuristics, or have introduced simplifying assumptions about the environment and type of computations in order to reduce complexity, e.g., [16] where all tasks are assumed to be of the same length, or the presence of only two computing nodes. HEFT [5] is one of the most popular heuristics, with an outcome comparable to alternatives, as detailed in [16]. HEFT has largely been used

in static environments, with a few exceptions. E.g., [17] uses HEFT and reschedules when a resource monitor discovers a new compute node, but does not consider node failures and assumes compute nodes dedicated to the scheduler, both of which are not realistic in tactical networks. In [18], HEFT is also used for re-scheduling, but it is only performed at carefully selected points during the execution.

Additional prior work in the area of fault tolerant scheduling of TGs includes the offline approaches [19], or online approaches with limited dynamic decision making [20]. However, they employ considerable task duplication to maximize the computation survival probability, precluding them for resource-constrained tactical MANETs. Other fault tolerance approaches include preserving task inputs [21]; checkpointing of the current execution frontier of the data-flow graph performed in parallel with the computation once the data has been created [22]; or message-driven execution and message logging for fault recovery [23]. However, they have not been applied to dynamically changing networking environments.

*Task completion time estimation*: A large body of prior work has focused on prediction of performance interference of computational tasks on CPUs and GPUs in the data center settings [24]–[27]. Interference considered in multi-core CPUs includes shared cache, main memory and function units for SMT [24], [25]. Non-preemptive accelerators (e.g., GPUs), however, have different sources of resource contention and exhibit different behavior [26], [27]. Existing TCT estimation approaches assume that the execution profiles of *all* competing tasks. We, however, aim to estimate TCT when unknown background tasks compete with profiled tasks.

*Path available bandwidth estimation*: Due to space limitations, we refer the reader to the related work sections in [14].

## V. Conclusion and future work

We presented the design and initial implementation and evaluation of Tactical Jupiter, a dynamic scheduling system for dispersed computing applications expressed as task graphs, that addresses multiple challenges posed by tactical networks. Evaluated in the field at NetModX'21, Tactical Jupiter is currently in transition to the US Army.

Our next steps include improving system robustness (including mapper election without restarting the task graph, already enabled by our design), extending TCT estimation to variable task output sizes, tuning parameters, and investigating incorporation of dynamic task graph scheduling into Kubernetes.

## References


[1] Y. Kim, C. Song, H. Han, H. Jung, and S. Kang, "Collaborative task scheduling for iot-assisted edge computing," *IEEE Access*, vol. 8, pp. 216 593–216 606, 2020.

[2] Y. Gao, W. Wu, H. Nan, Y. Sun, and P. Si, "Deep reinforcement learning based task scheduling in mobile blockchain for iot applications," in *ICC 2020*, 2020, pp. 1–7.

[3] P. Ghosh, Q. Nguyen, and B. Krishnamachari, "Container Orchestration for Dispersed Computing," in *Proc. 5th International Workshop on Container Technologies and Container Clouds*, December 2019.

[4] P. Ghosh, Q. Nguyen, P. K. Sakulkar, A. Knezevic, J. A. Tran, J. Wang, Z. Lin, B. Krishnamachari, M. Annavaram, and S. Avestimehr, "Jupiter: A Networked Computing Architecture," arXiv preprint arXiv:1912.10643, 2019.

[5] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, 2002.

[6] S. Li and S. Avestimehr, *Coded Computing: Mitigating Fundamental Bottlenecks in Large-scale Distributed Computing and Machine Learning*. Now Publishing, 2020.

[7] P. Sakulkar, P. Ghosh, A. Knezevic, J. Wang, Q. Nguyen, J. Tran, H. K. G. Narra, Z. Lin, S. Li, M. Yu, B. K. ans S. Avestimehr, and M. Annavaram, "Technical Report ANRG-2018-01: WAVE: A Distributed Scheduling Framework for Dispersed Computing," University of Southern California, Tech. Rep., 2018.

[8] R. Wolski, N. Spring, and J. Hayes, "Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid," in *Proc. 8th International Symposium on HPDC*, 1999.

[9] K. S. Hasan, J. K. Antonio, and S. Radhakrishnan, "A New Multi-core CPU Resource Availability Prediction Model for Concurrent Processes," in *Proc. IAENG International Conference on Computer Science*, 2017.

[10] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, 1997.

[11] Phoronix, "The Phoronix Test Suite: Linux Testing and Benchmarking Platform," https://www.phoronix-test-suite.com, 2021.

[12] H. Shen and X. Hong, "Host Load Prediction with Bi-directional Long Short-Term Memory in Cloud Computing," arXiv preprint arXiv:2007.15582, 2020.

[13] T. Clausen, C. Dearlove, P. Jacquet, and U. Herberg, "The Optimized Link State Routing Protocol Version 2," RFC 7181, 2014.

[14] G. Kim, J. Lee, L. Kant, and A. Poylisher, "Utility-Driven Traffic Engineering via Joint Routing and Rate Control for 802.11 MANETs," in *Proc. COMSNETS'21*, 2021.

[15] R. Chadha, T. Bowen, C.-Y. J. Chiang, Y. M. Gottlieb, A. Poylisher, A. Sapello, C. Serban, S. Sugrim, G. Walther, L. Marvel, A. Newcomb, and J. Santos, "CyberVAN: A cyber security virtual assured network testbed," in *Proc. 2016 IEEE MILCOM*, 2016.

[16] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, 1999.

[17] Z. Yu and W. Shi, "An Adaptive Rescheduling Strategy for Grid Workflow Applications," in *Proc. IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[18] R. Sakellariou and H. Zhao, "A low-cost rescheduling policy for efficient mapping of workflows on grid systems," in *Proc. Sci. Prog. 12*, 2004.

[19] A. Benoit, M. Hakem, and Y. Robert, "Fault tolerant scheduling of precedence task graphs on heterogeneous platforms," in *Proc. IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[20] N. Tabbaa, R. Entezari-Maleki, and A. Movaghar, "A fault tolerant scheduling algorithm for dag applications in cluster environments," in *Proc. ICDIPC*, 2011.

[21] E. Maehle and F.-J. Markus, "Fault-tolerant dynamic task scheduling based on dataflow graphs," Fault-Tolerant PDS, 1998.

[22] N. Vrvilo, V. Sarkar, K. Knobe, and F. Schlimbach, "Implementing asynchronous checkpoint/restart for CnC," CnC, 2013.

[23] E. M. Rojas, "Scalable message-logging techniques for effective fault tolerance in hpc applications," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2013.

[24] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *Proc. 40th Annual ACM International Symposium on Computer Architecture (ISCA)*, 2013.

[25] Y. Zhang, M. Laurenzano, J. Mars, and L. Tang, "SMiTe: Precise QoS Prediction on Real System SMT Processors to Improve Utilization in Warehouse Scale Computers," in *Proc. 47th Annual ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[26] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads," in *Proc. 42nd Annual ACM International Symposium on Computer Architecture (ISCA)*, 2015.

[27] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers," in *Proc. 23rd ACM International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, 2017.