Delay/Disruption Tolerant Reinforcement Learning Aurora based Communication System (DREAMS)

Richard Stottler
Stottler Henke Associates, Inc., San Mateo, CA 94002
Gregory Howe
Stottler Henke Associates, Inc., San Mateo, CA 94002

ABSTRACT

Future satellite networks necessitate new network traffic routing algorithms that can route large volumes of network traffic in networks characterized by intermittent links, high latencies, low bandwidths, ad hoc connections, high bit error rates and diverse nodes with resource and computational abilities; furthermore, these routing algorithms need to be executed on resource constrained nodes and be capable of utilizing multi-hop communications. To this end, we have developed the Delay/disruption tolerant REinforcement learning and Aurora based coMmunication System (DREAMS), a distributed, near optimal link optimization and routing algorithm. Here, link optimization refers to the ability to optimize a given link by tuning software defined radio parameters to maximize link quality metrics such as bandwidth, power consumption and bit error rate, while routing refers to scheduling storage and transmission of data to maximize network throughput. DREAMS consists of three primary components: 1) a Machine Learning-based link optimizer which periodically optimizes each network link through tuning parameters for software defined radio and apprises the distributed schedulers of what data bandwidth each link can support; 2) a packet predictor which predicts how many specific future packets are expected from specific applications (based on a daily activity schedule) and the expected volume of packets not tied to the schedule; and 3) the distributed schedulers themselves, which exchange network status and schedule information with one another to continuously update and re-optimize the transmission and storage schedule. By rescheduling frequently, these distributed schedulers react quickly and in an optimal manner in response to unexpected events such as links or nodes going down or the sudden arrival of unexpectedly high volumes of data packets.

1. INTRODUCTION

NASA's Space Communication and Navigation (SCaN) program is a NASA effort to build, maintain, and expand a scalable, integrated, flexible, comprehensive, robust, cost effective, and efficient space communications infrastructure that provides communications capabilities in the face of an unpredictable environment. SCaN integrates with and manages operations on NASA's three main networks: the Deep Space Network (DSN), the Near Earth Network (NEN), and the Space Network (SN), all of which operate continuously. The DSN consists of three ground stations that communicate with deep space satellites. The NEN offers support in a variety of frequency bands to various sub-orbital, LEO, MEO, HEO, lunar, and Lagrange orbits. The SN consists of some LEOs, a ground relay system, and ten geosynchronous Tracking Data Relay Satellites (TDRSs). DSN, NEN, and SN ground stations and satellites as well as other spacecraft (e.g., the lunar Gateway, lunar relay satellites (LRSs), cubesats, and surface assets (e.g., science stations, surface relays, and astronauts) may all serve as communications originators, endpoints, relays, and/or stores.

The vast distances and numerous satellites in current and future NASA missions necessitate multi-hop communications utilizing temporary stores. Space communication networks are often characterized by intermittent links, high latencies, low bandwidths, ad hoc connections, mobile physical nodes, asymmetric data rates, higher error rates, heterogenous node types (e.g., International Space Station (ISS) versus an LRS). All these characteristics are variable (may depend on the time, the node, etc.) and may have a predictable aspect and an unpredictable aspect (e.g., line-of-sight (LOS)-based network disruption is predictable, but sporadic space weather is not).

NASA spacecraft and ground stations must determine how best to maximize Quality of Service (QoS) and bandwidth (including using channels when they are not used by other NASA or non-NASA users) in a constantly changing space environment (e.g., solar flares, solar wind, Coronal Mass Ejections (CMEs)). Spacecraft must minimize the use of scarce resources such as power (including both overall spacecraft power and Tx power), bandwidth, time, and Tx- and Rx-related resources such as allowed frequency bands and multiple Tx/Rx phased

array elements. Spacecraft and surface assets must consider additional constraints on these resources (e.g., simultaneous Tx/Rx capabilities).

With the advent of Software-Defined Radios (SDRs) and the upcoming missions to the Moon, Mars, and beyond, cognitive technology is needed to maximize throughput in the face of volatile conditions (including space weather and onboard equipment failures) and potentially long latencies. The SCaN testbed on the ISS is currently testing three third generation SDRs, and the Mars Science Laboratory (MSL) is equipped with a pair of second generation SDRs. In the future, more and more missions will utilize SDRs. SDRs offer customizable parameters that cognitive technology can manipulate (e.g., transmit frequency, modulation scheme) to maximize throughput, minimize Bit Error Rate (BER), and optimize other measures (e.g., power efficiency, spectral efficiency).

In this paper, we will present the Delay/disruption tolerant REinforcement learning and Aurora based coMmunication System (DREAMS), a scheduling algorithm that addresses the issues inherent to communications between satellites and exploits the configurability of SDRs. The DREAMS system consist of three primary components: 1) a ML-based RF link optimizer which periodically optimizes each RF link and apprises the distributed schedulers of what data bandwidth each link can support; 2) a packet predictor which predicts for the distributed schedulers both how many specific future packets are expected from specific applications (based on the daily activity schedule) and the expected volume of additional packets not tied to the schedule; and 3) the distributed schedulers themselves, which exchange network status and schedule information with one another to continuously update and re-optimize the transmission and storage schedule.

2. DREAMS SYSTEM OVERVIEW

DREAMS on One Node

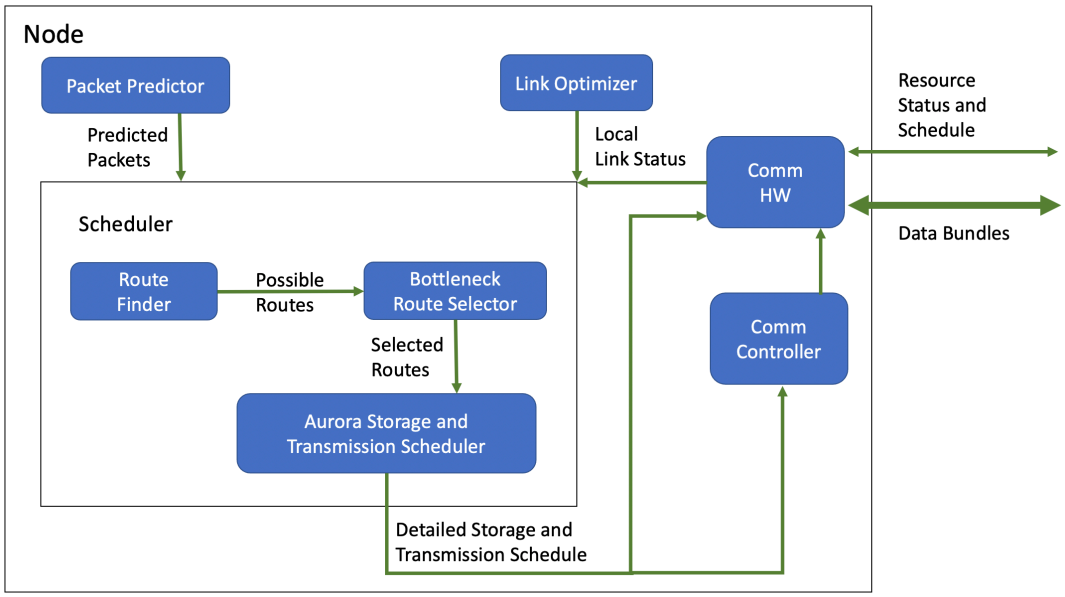

Fig. 1. DREAMS on One Node.

Fig. 1 shows the architecture for the DREAMS agent that resides on each node.

The Packet Predictor gives predictions for packets associated with both scheduled events and unscheduled events. These are summed together to form a prediction of packets in general. This is done for each pair of nodes (source & destination) and includes all the major properties of a packet (source, destination, origination time, deadline, size, priority).

The Link Optimizer outputs suggestions for parameters to optimize links between nodes and the quality of the link for each set of recommended parameters (quality of the links refers mainly to bit rate but also to metrics like BER which are useful for deciding how large bundles should be, how large "super-bundles" should be (described later), how to optimize transport layer parameters, etc.).

The main inputs to the scheduler are the actual packets that have arrived and need to be scheduled/routed, the predicted packets that will arrive in the future that can be optionally scheduled/routed in advance, the status of the links and nodes in the network, including the optimized data bandwidth of each link, and the most recent overall schedule from neighboring nodes (nodes with a direct communication link to this one). Each node does not schedule the entire network, but only tackles a specific subset of the problem. Specifically, each node schedules and routes the bundles that first arrive at it (i.e., it is the origination node for the bundle) and that are scheduled to pass through it on the bundles' currently scheduled route (the latter case to provide the ability to change a bundle's route due to changed conditions in the network).

A first step is to update the predicted visibilities, then run Viz-Dijkstra's algorithm for each bundle to find a set of feasible routes for each bundle (i.e., all routes meet the bundle's deadline) that include both fastest routes and least congested routes (that still meet the deadline). Congestion can be determined by a variety of means, including current traffic through each node and link, currently scheduled traffic on each node and link as given by the most recent schedule possibly including scheduled predicted bundles that have not yet arrived and scheduled predicted traffic volume (that are not necessarily specific bundles but are more of a bytes/sec description between different origins and destination), using the first "probabilistic allocation" of the Bottleneck Avoidance Algorithm Route Selector, or a combination of the above. The bundle metadata (origin, destination, size, deadline, priority, etc.) and the set of possible routes for every bundle are passed to the Bottleneck Avoidance Algorithm Route Selector.

The first step for the Bottleneck Route Selector is to register traffic scheduled by other nodes (and which will not be rescheduled by this node because the bundles in question are not scheduled to pass through it). This is merely adding the transmit and storage times and volumes to the antennas, links, and node objects in the Bottleneck Route Selector. Then the Bottleneck Route Selector must perform its initial step, to "probabilistically allocate" each bundle's resource requirements across all of the resources on the potential routes. For example, if there are 10 possible routes, each of the ten routes will have allocated to the nodes and links on the route 10% of the storage, link bandwidth, and antenna resources that the bundle actually needs. (E.g., 10% of the memory storage actually needed by the bundle at each node on the route, 10% of each send and receive antenna on the route, and 10% of the transmission bandwidth.) These are added up across all bundles (and on top of the already scheduled traffic that was previously registered) across all resources. This is the first step of the flow chart given above in Fig. 6. Then, as shown in the flow chart, as long as there are bundles left to select a route for, the system finds the maximum peak across all times and resources and the task (such as send, store, or receive) that most contributes to the peak (not counting tasks already scheduled) which normally means the task with the least number of alternatives, and then schedules that task and the overall flow (i.e., route) that it is a part of. To schedule the overall route means to first remove all the "probabilistic allocations" for all of the tasks associated with that route, then make resource and time assignments (that meet the bundles deadlines) to the specific tasks of the route that minimize the peaks overall. This typically means making selections from the "valleys" in the resource profiles. Note, as described above, this algorithm tends to "spread out" bundle traffic (to the degree possible to still meet the deadlines) across all links, especially otherwise-under-utilized link to improve overall network throughput. When all bundles have been scheduled, then this step is done and the bundle meta data and chosen route for each bundle are passed to the detailed Aurora Transmission and Storage Scheduler.

Given the specific route for each bundle (and the previously scheduled bundles from other nodes), the Aurora Transmission and Storage Scheduler has the relatively straightforward job of determining the precise beginning and ending of each send, transit, receive, and storage task to ensure that nothing is overallocated or over capacity and to also minimize configuration change or antenna slewing time. It typically does these two things by sorting the bundles first by which next node each is going to (so bundles going to the same next node appear together) and secondarily by the transmission deadline needed to make the visibility opportunity to meet the bundle's ultimate deadline. Then these groups are sliced and intermingled to minimize antenna link switching time while still meeting deadlines. If the antenna link switching time tends to be very significant, a factor can be added to the route selector

to tend to select the same or overlapping routes for bundles that are transiting the network at about the same time to maximize the opportunities for the Aurora Transmission and Storage Scheduler to minimize antenna link transition time. If there is more traffic requested that can transit through the network in time to meet deadlines, even if only true for some links or nodes, Aurora will also have the function of making sure that the highest-priority bundles get scheduled, even if that meets having to swap out some lower-priority bundles that were previously scheduled. As described previously, the use of priorities last, instead of first allows substantially more optimization (typically 15 to 20% more throughput) while still obeying the requirements of the priorities.

Distributed DREAMS

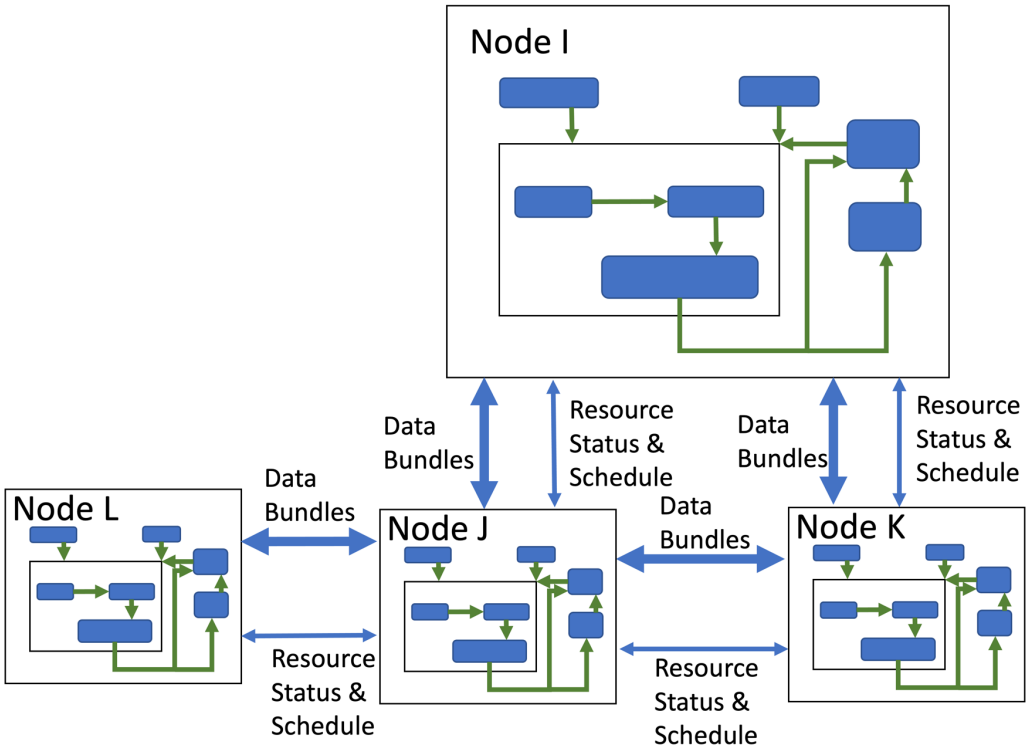

Fig. 2. Distributed DREAMS Diagram.

As shown above, DREAMS can be modified to be a distributed scheduling system. To conform to our proofs of correctness, termination, and that data packets do not avoidably being cycle back to the same node after being sent from that node previously, the distributed DREAMS must obey a few rules:

- Only Nodes on a packet's route schedule/reschedule it (as described above).
- Schedule data is much smaller in volume than the Data Packets (as shown in this Final Report, the Schedule Data is about 0.2% of the volume of the data packets.
- The schedule/route for a packet arrives at a node before the packet does.

Although not absolutely necessary to fulfill the third requirement, we would recommend that a separate (low volume) channel exist for schedule and network resource status data. This will ensure that the schedule/route for a packet arrives before the packet does but will also ensure that information on major impacts to the network (e.g., a node or link going down, a flood of high-volume, high-priority, quick-deadline traffic, etc.) propagate most rapidly through the network so it can be reacted to with maximum speed. A final note is that each node will only need to schedule/reschedule a fraction of the bundle traffic on the network, lowering its processing time.

3. DREAMS TRAFFIC PREDICTION

There are two elements to the packet predictor. The first is predicting packets associated with scheduled events. The second is predicting packets (or at least rough estimates of data traffic) that are unscheduled. We implemented the predictor for scheduled traffic but were not able to train and validate it. We will be acquiring data (real and simulated) to train the already implemented network traffic prediction algorithm described below.

DREAMS will predict network traffic between nodes given scheduled events. This predicted network traffic is used to create placeholder network traffic that is passed to the DREAMS scheduling and routing algorithm. In essence, by adding the placeholder traffic, DREAMS avoids using visibility windows that are needed for high- priority scheduled transmissions. Currently, our packet prediction is limited to predicting scheduled transmissions.

We will train a neural network to predict meta information about the event from the scheduled event with our existing pipeline. Specifically, the training pipeline works in three stages: First, it takes in a scheduled event, $e^{(i)}$. $e^{(i)}$ is mined for features including the scheduled start time, scheduled end time, even type (phone call, video call, image upload, etc.), source station, and target station. Call the feature vector $x^{(i)}$. Second, the pipeline extracts a label from all the groups of packets that were associated with the scheduled event. The label for an event is $y^{(i)} = (y_s^{(i)}, y_d^{(i)}, y_o^{(i)}, y_t^{(i)}, y_p^{(i)})$ where $y_s^{(i)}$ is the sum of the size of the packets, $y_d^{(i)}$ is the delay or difference between when the event is scheduled to start and when the first packet of the event is created, $y_o^{(i)}$ is the over time or difference between when the event is scheduled to end and when the last packet of the event is created, $y_t^{(i)}$ is the time to live or average difference between when a packet is created and its deadline, and $y_p^{(i)}$ is the priority of the event. Lastly, we train a neural network f_θ to minimize a weighted sum of squared errors. Specifically, we train f_θ to minimize

$$\sum_{i=1}^{n} (y^{(i)} - f_\theta(x^{(i)}))^T \Lambda (y^{(i)} - f_\theta(x^{(i)}))$$

Where Λ is a positive diagonal matrix with entries corresponding to relative importance weightings between the predictions, e.g., if $\Lambda_{11} = 1$ and $\Lambda_{22} = 2$ this would penalize the network more for mis-predicting the start time of the event more than the size of the event. More traditional statistics methods are possible here, but we implemented a neural network due to its versatility and ability to adapt to more complex losses in the future, e.g., if we wanted to predict categorical information such as source and destination.

When deployed, the packet prediction module uses a separate three stage pipeline. First, it takes in a scheduled event, $e^{(i)}$. $e^{(i)}$ is mined for a feature vector, $x^{(i)}$. Second, we use our trained neural network to make a prediction $\hat{y}^{(i)} = (\hat{y}_s^{(i)}, \hat{y}_d^{(i)}, \hat{y}_o^{(i)}, \hat{y}_t^{(i)}, \hat{y}_p^{(i)}) = f_\theta(x^{(i)})$. Lastly, we process this prediction to create filler-packet groups for the network. Specifically, we find the predicted start $t_s^{(i)}$ and end times $t_e^{(i)}$ by adding the delay time, $\hat{y}_d^{(i)}$, to the scheduled start time and adding the over time, $\hat{y}_o^{(i)}$, to the scheduled end time. We set the predicted number of packets, $n^{(i)}$, to be the size of the event $y_s^{(i)}$ divided by a tunable parameter for the average size of groups of packets for DREAMS. Finally, we create $n^{(i)}$ packet groups in equal time intervals between $t_s^{(i)}$ and $t_e^{(i)}$ with a deadline $\hat{y}_t^{(i)}$ after they are created. These groups are all assigned the same priority $\hat{y}_p^{(i)}$ and are written to a json file, which is used by DREAMS to allocate placeholders in the scheduler.

There are few ways to improve our scheduled packet prediction. First, we're currently working with simulated network traffic and scheduled events. With real data, we will be able to train our network to predict packets more accurately for real, scheduled events. With access to subject matter experts, we will be able to further add and refine the features we use in our predictor; for example, lunar position. We can experiment with predicting more detailed information about each event that could either be useful to the scheduling algorithm or NASA. Lastly, we can try to apply more data-efficient traditional statistics algorithms to the task and measure how they perform against our neural network in real-world settings.

We are adding the ability to predict packets that don't correspond to any scheduled events. This can be viewed as predicting the background network traffic between stations after removing all the packets belonging to scheduled events. We plan on trying and comparing a handful of methods to do this.

First, we plan on implementing a neural network that uses the source location, the target location, and other features that we've had success with such as time of day, time of year, proximity to holiday, space weather, etc. to predict the aggregated (over time) background network traffic generated at the source location that will be transmitted to the target location. The details for training and prediction when deployed are identical to those of scheduled packet prediction with one minor difference. The aggregation will be over time instead of events, e.g., the network will predict the total traffic (and meta data) for packets generated at a source that will be transmitted to a target during a specific time interval. This method has a few major benefits. First, its input size scales linearly as we add more nodes to the network. Second, it is data efficient. If we trained a network for each individual source target pair, for most pairs (except between hubs), the data would be very limited, which would result in overfitting or needing to use smaller models. The proposed architecture should be able to leverage relationships learned from data from different source, target pairs. Specifically, given two source target node pairs (s, t), (u, v), the neural network approach can leverage relationships learned from traffic between s and t to predict better on traffic between u and v. Third, the model can be trained/evaluated on select pairs of inputs, e.g., doesn't need to compute traffic for all node pairs. Fourth, the model can be trained on any differentiable user-specified loss, which is often not the case for traditional statistical time series models. Finally, the model can learn complicated non-linear relationships between our features.

Second, we plan on implementing and comparing our neural network approach to more traditional statistical models for time series data such as Prophet [1]. The Prophet model lacks the flexibility to train one model and use the same model for multiple nodes, so we will need to train a distinct model for each source target pair that we want to predict on. However, it should be a good interpretable baseline to compare to our neural network-based method.

2. LINK OPTIMIZATION APPROACH

KRATOS has lent us QuantumRadio and SpectralNet. The QuantumRadio + SpectralNet package is referred to as the KRATOS Simulator. The KRATOS Simulator has many capabilities including the ability to modify modulation schemes, coding schemes, power, roll-off, frequency, symbol rate, delay, doppler effect, matched filters, noise level, and potentially multipath effects. The KRATOS Simulator also gives metrics such as energy per bit to noise power spectral density ratio (E_b/N_0), energy per symbol to noise power spectral density ratio (E_s/N_0), and bit error rate (BER) that we are using to train RL based link optimization agents.

Deep Q-Networks (DQNs) [2], a deep reinforcement learning technique, has been successfully applied to optimize communication links between satellites by varying parameters such as modulation scheme, encoding rate, symbol energy and rate, and roll-off factor. Although this technique has shown promising results and can handle continuous state spaces, it assumes a discretized action space, which is typically not the case in link optimization—encoding rate, symbol energy, and rate are not discrete. This is usually addressed by discretizing each continuous sub-action e.g., discretizing the symbol rate. For example, consider an action a, which takes values in an interval $[0, 1]$, e.g., the percentage of our maximum transmit power that we want to use. This action is necessarily continuous but can be approximated with discrete actions, e.g., set power to 0.25 or 0.75. DQNs discretize in this fashion, which often loses finer-grained control over continuous actions. Additionally, DQNs require an output node for every possible combination of actions—more precisely, every possible combination of actions that you'd want to consider. This results in an exponentially increasing action space as we add additional parameters; for example, if we later wanted to add the ability to specify an encoding scheme where we have 5 possible values, this addition would result in needing 5 times more output nodes. These issues are either solved or reduced by a newer method called Deep Deterministic Policy Gradients (DDPG) [3]. Specifically, DDPG can propose continuous actions, which results in finer-grained action control and removes the exponential increase in output nodes.

At a high level, DDPG trains an actor neural network to propose actions or parameters to optimize a 'reward' across a series of interactions with the environment; this reward function (over a sequence of actions) is approximated by a critic network. For link optimization, the reward is typically some combination of low transmit power, low bit error

rate, and high bit rate. Note that DDPG allows for this reward to be fully customizable and does not require explicit knowledge of how to calculate the reward from a state and action pair and instead can use external factors related to the environment. More formally, DDPG uses the deterministic Bellman Equation

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1}}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1};\, \theta^{\mu}))]$$

Where μ maps a state (this could involve space weather, previous parameter configurations, distance between the satellites, etc.) to an action; this is typically implemented as a neural network with learned parameters θ^{μ}. r is a reward function that takes in a state and action and outputs a reward value. Note that r is typically not a function that we know explicitly and usually involves interacting with the environment, e.g., broadcasting over a link. $\gamma \in [0,1]$ is a discounting factor that specifies how much emphasis to place on the current reward instead of the reward over a future sequence of actions performed in different states at different times. Lastly, Q^{μ} is the Q function, which evaluates the expected sum of (weighted by the discount factor) rewards of performing action a_t in state s_t and using the actor μ to select actions for the remainder of the trajectory (in our scenario, a trajectory is a sequence of state action pairs from the start of a visibility window to the end of a visibility window where states and actions may both be dependent on the actor μ). Q^{μ} is typically not known ahead of time, so it is approximated by another neural network specified by parameters θ^{Q}, which are optimized by minimizing the temporal difference error:

$$\mathbb{E}_{s_t, a_t, r_t}\left[(Q(s_t, a_t\,;\, \theta^{Q}) - (r(s_t, a_t) + \gamma Q(s_t, a_t\,;\, \theta^{Q})))^2\right]$$

Simultaneously, the actor network $\mu(\cdot\,;\, \theta^{\mu})$ is updated by taking steps according to the policy gradient

$$\mathbb{E}_{s_t}\left[\nabla_a Q(s_t, a_t\,;\, \theta^{Q})|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^{\mu}} \mu(s_t\,;\, \theta^{\mu})|_{s=s_t}\right]$$

These updates are applied on every iteration of the DDPG algorithm. At a high level, the first step makes the critic better at guessing how good an action performed given a state is while the second step makes the actor better at producing actions that the critic thinks are good. If the critic is accurate at gauging the quality of actions and the actor is good at producing actions that the critic thinks are good, then the actor is outputting actions that result in high rewards in the environment.

We've set up the KRATOS Simulator to supply an OpenAI Gym environment to train our agent. Through API calls to the KRATOS Simulator, the simulator supplies us with state information such as the relative velocity and relative acceleration, and results from our last decision, e.g., what BER and E_b/N_0 we achieved during the previous step and the configuration/action the agent proposed during the previous step. This state information is then synchronized between the transmitting and receiving agent, which is accomplished by waiting for the information to propagate to both satellites. Once the information has been synchronized between both agents, we calculate the same action (an action is composed of transmission configuration parameters such as modulation scheme, matched filter type, and matched filter rolloff) at both the transmitting and receiving agent, which is possible due to waiting to synchronize the state between agents, select that action and apply those transmission parameters to the receive and transmit antennas through an API call for the next few seconds until a new state is generated.

We define our reward function to be a linear combination of the E_b/N_0, BER, and Lock Losses and are training a multitude of agents with different tradeoffs between BER and bandwidth. Specifically, our reward is

$$\lambda_1\,(1 - BER)^{\alpha} - \lambda_2\, LockLosses + \lambda_3\, \mathrm{Eb/N0}$$

Where $\lambda_1, \lambda_2, \lambda_3$ define relative weightings between the loss terms and $\alpha > 0$ is a sharpening constant used to modify the emphasis placed on achieving BER's very close to 1 and LockLosses is the count of times the lock between the satellites is lost. In effect, by prioritizing different terms we can modify what the agent prioritizes optimizing during training.

When deployed, our link optimization agent can both evaluate the quality of any link by evaluating the Q value of the initial state (before any transmission between the pair has occurred) and choose transmission configuration parameters to optimize the communication between any two satellites.

3. ROUTE FINDING

The Route Finder finds good routes for each bundle. At a high level, a better route is one that ends earlier in time and/or avoids congestion. To find the single fastest route (as well as to enable finding fast and uncongested routes),

we created Viz-Dijkstra's Algorithm, based on Dijkstra's Algorithm. To explain Viz-Dijkstra completely, we show how to model the DTN as a graph, how to frame the problem as a type of path-finding problem, how Dijkstra operates, and finally how Viz-Dijkstra operates.

First, the DTN should be converted into a time-dependent graph where each edge's "availability" (when the edge exists) and weight can change with time. Each DTN node should be treated as a node in the graph. Now, for every pair of nodes, there is a time-dependent edge. For this discussion, we will use "aggregate edge" to mean the edge across all time and "edge" or "visibility window" to refer to a specific period of time for an aggregate edge when it is available. For an aggregate edge between A and B (assume symmetric links for now), that aggregate edge's availability equals the periods of time when A and B can communicate (i.e., there is a line of sight and the two nodes' antennas and protocols are compatible). Each period of availability will be called a "visibility window." For each visibility window, the edge's weight is the bandwidth (bits per second).

We define metadata for each bundle that we want to transmit. The metadata includes the source node, destination node, the size, the origination time, and the deadline of the bundle (when the bundle needs to be received by the destination node (we discuss multiple destination nodes later)). Calculating a route for a single bundle (ignoring all others for the time being) is equivalent to finding a valid path in this graph.

As a starting point, we discuss Dijkstra's algorithm [4], hereafter called "Dijkstra," which is a commonly used algorithm for finding the shortest (or lowest cost) path between a source and a target vertex in a directed graph with positively weighted edges with no time dependency. Dijkstra maintains a set of the closest vertices to the source vertex, and the minimum cost to reach each of these closest vertices from the source vertex. Dijkstra finds the next closest vertex by considering for each outgoing edge $e(v_i, v_j)$ from one of the closest vertices v_i to an unvisited vertex v_j calculating a cost $c_{ij} = c(v_i) + c(e(v_i, v_j))$ where $c(v_i)$ is the lowest cost to reach vertex v_i and $c(e(v_i, v_j))$ is the cost to use edge $e(v_i, v_j)$. The smallest pair $(r, s) = argmin_{ij} c_{ij}$ is chosen, and v_s is added to the closest vertices set as the next closest vertex to the source with minimum cost to reach of $c(v_s) = c(v_r) + c\big(e(v_r, v_s)\big)$. Note that in implementation, Dijkstra computes $(r, s) = \text{argmin}_{ij} c_{ij}$ efficiently at each iteration by pulling from a priority queue of c_{ij}s maintained across iterations. This process is repeated until every vertex has been discovered. If every edge cost $c(e(v_i, v_j))$ is nonnegative, then each $c(v_i)$ is guaranteed to be the lowest possible cost to reach vertex v_i. This algorithm runs in $O(|E| \log |E|)$ where $|E|$ is the number of edges and through back-tracking on the computed vertex costs can be used to reconstruct the lowest cost route between the source node and any other node. Dijkstra's algorithm doesn't encode time-dependent information about the edges e.g., when satellites can and cannot see each other, so it needs to be modified to be applied to DREAMS.

We define a visibility window to represent a single time interval when a node (node1) can send data to another node (node2) with the following characteristics: an upper bound on the distance between the two nodes during the visibility time interval, how congested the window is (how much traffic—this of course depends on our routing decisions and is not inherent to the graph), and the bandwidth of the link between the two nodes. For each node1 to node2 link, we do not assume that the reverse link from node2 to node1 also exists (upload/download speeds can be asymmetric), but we *do* assume that there is sufficient bandwidth from node2 to node1 to send a received acknowledgement (ACK) when required by the transmission protocol. From a visibility window, we calculate one main quantity of interest, the earliest possible arrival time of a bundle using the visibility window. Specifically, given a metadata object p and the current time t_c, we calculate the bundle's arrival time at node2, t_{wa}, across the visibility window, w, as $t_{wa} = \max(t_c, t_{ws}) + \frac{d_w}{c} + \left(\frac{d_w}{c} + \frac{s_p}{b_w}\right)$ where d_w is an upper bound on the distance between the two satellites, c is the speed of light, s_p is the size of the transmission (e.g., in megabits), t_{ws} is the start time of the visibility window, and b_w is the bandwidth of the link. We allocate an additional $\frac{d_w}{c}$ factor to allow time for an acknowledgement to be returned from node2 to node1 after the completion of the transmission. We take $\max(t_c, t_{ws})$ to reflect needing to wait for the visibility window to start. If the calculated arrival time t_{wa} is greater than the end time of the visibility window t_{we} we change t_{va} to be infinity to reflect that the transmission is impossible during the visibility window.

Given a list of visibility windows between all nodes in a network and a metadata object corresponding to a bundle, we propose N 'fast,' M 'low congestion,' and B 'balanced' route (for a total of R=N+M+B routes) for the bundle. The R is a maximum, and there may be many cases where fewer than R routes will be returned. N, M, and B are adjustable parameters; we used N=M=5 and B=1. We use a visibility aware variant of Dijkstra to construct these routes, which we will call Viz-Dijkstra. Viz-Dijkstra's algorithm can be interpreted as adjusting the edge cost function $c(e(v_i, v_j))$ to be dependent on the arrival time. Specifically, we make the following changes: Each edge - $e(v_i, v_j)$ is now stored as a set of all visibility windows w between vertices v_i and v_j. We create a new time-dependent loss/cost function for using a visibility window $c_w(t_c) = \alpha(t_{wa} - t_c) + (1 - \alpha)g_w$ where g_w is a non-negative constant indicating how congested the window is and $\alpha \in [0,1]$ is a tunable hyperparameter which specifies the importance of arrival time versus congested resource usage (note in our implementation that we set this cost to be infinity if t_{wa} occurs after the bundle's deadline). We find the lowest cost visibility window $w^* = \text{argmin}_{w \in e(v_i,v_j)} c_w(t_{c_i})$ and define the new cost function to be $c(e(v_i, v_j), t_{c_i}) = c_{w^*}(t_{c_i})$. We also change the calculation for the cost c_{ij} to be $c_{ij} = c(v_i) + c(e(v_i, v_j), t_{c_i})$. To reflect the need to know the time, we reach v_i to be able to compute c_{ij}, and whenever we find the low-cost path to v_i, we also store the arrival time t_c. Lastly, we store the visibility window, w_i^* that we used to reach v_i, so we can efficiently compute the lowest-cost path from the source vertex to v_i. These changes allow us to use Dijkstra's algorithm to compute low-cost paths. Our modified Viz-Dijkstra algorithm runs in $O(|W| + |E| \log |E|)$ where $|W|$ is the number of visibility windows in the graph and $|E|$ is the number of nodes that have at least one visibility window between each other.

We find N 'fast,' M 'low congestion,' and B 'balanced' routes for a bundle using the following three procedures. For the 'fast' routes, the algorithm uses Viz-Dijkstra with $\alpha = 1$ ($\alpha = 1$ places all the cost on time) to select the fastest possible route from the source node to the target node. The most congested visibility window on this route is removed from the graph and this process is repeated (N-1) more times. This procedure produces N 'fast' routes with varying congestion. The M 'low congestion' routes are produced similarly. Specifically, the algorithm uses Viz-Dijkstra with $\alpha = 0$ ($\alpha = 0$ places all the cost on congestion) to select a low congestion possible route from the source node to the target node. The most time-consuming visibility window used on the route is removed from the graph, and the process is repeated (M-1) more times. The B 'balanced' routes are only relevant if no 'low congestion' routes were found. We produce 'balanced' routes by using a binary-searching to find low values of α that still return feasible routes (routes that arrive by the deadline for the bundle). All routes returned by this procedure are guaranteed to be valid routes that reach the destination before the deadline.

So far, we have only discussed the case when a bundle has a single destination node. In the BP, endpoint IDs specify an endpoint (note that these are not hierarchical like in IP). Endpoints are a non-empty set of nodes. A bundle's source and destination are each an endpoint. Each node is part of at least one endpoint. For each node, it is part of a singleton endpoint (a set consisting of exactly that one node), and it can optionally be part of additional, non-singleton endpoints that each consist of multiple nodes. "Multiple destinations" is equivalent to stating that the destination endpoint is not a singleton set. There are two delivery "modes," which we call ONE mode (delivery to any one node in the endpoint is considered a successful delivery) or ALL mode (successful delivery means the bundle must be delivered to all nodes in the endpoint). Handling the ONE mode is straightforward and included in the algorithm. Viz-Dijkstra is run with a target *set* of nodes instead of a single target node, and the iteration stops when any node in the set is reached. Handling the ALL mode will be implemented in our future work.

In conclusion, for each bundle, we generate at most (since some low congestion routes may be duplicates of fastest routes) R routes using the above procedure. All the proposed routes are passed to our bottleneck avoidance scheduling algorithm, which selects the one route that will tend to produce the best result in the overall schedule while still meeting the needs of the individual bundle. Specifically, it chooses the route that tends to reduce contention and congestion overall while still meeting the bundle's deadline. Providing the scheduling algorithm with R potential routes for each bundle offers the flexibility to choose a route that will not block the transmissions of the other bundles.

4. SCHEDULING SIMULATION

We mimicked to a reasonable extent a small-scale version of the future LunaNet. We used Orekit with some post-processing to generate orbits using orbital parameters similar to actual and proposed satellites. For example, Fig. 3 shows the orbits we used for our simulated TDRS-K, TDRS-L, and TDRS-M satellites, and Fig. 4 shows the three orbits for the five Ideal Lunar Relay Satellites we used (two of the orbits have two satellites each, which is described in the following paragraph). Note that this does not mean that other nodes (e.g., the Lunar Gateway, ISS) cannot serve as relays, but rather that these five are envisioned to serve primarily as relays. Relays can also be originators and destination nodes (e.g., for exchanging PNT information), but originating and "destinating" are not their primary purposes. We used link latencies that were calculated from the actual distance between the two nodes on that link.

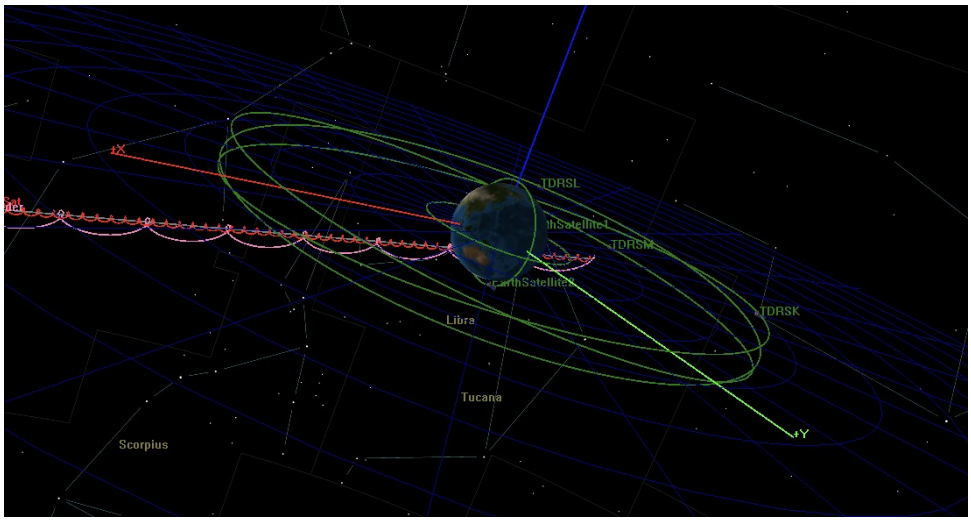

Fig. 3. TDRS-like Orbits.

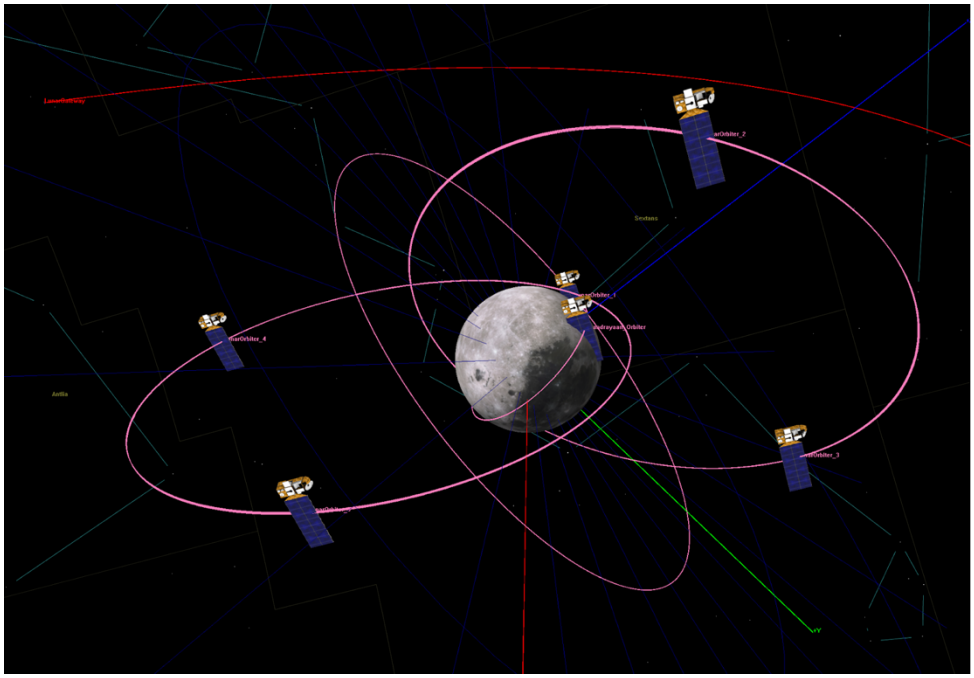

Fig. 4. Simulated Ideal Lunar Relay Satellite and Chandrayaan Orbiter Orbits.

The circular equatorial orbit contains one satellite. The two elliptical orbits contain two satellites each with different anomalies (we used "opposite" anomalies, i.e., 180 degrees apart). This provides good coverage of the lunar south pole while also maintaining good cross links between the relays. This constellation is incremental (one satellite can be launched at a time), scalable (more can be added later), and provides good coverage (of the south pole and for

cross links between relays, with long visibilities and minimal gaps). This is an IOAG recommendation. The IOAG ideal orbits are shown in Fig. 5. Table 1 shows the IOAG recommended orbital parameters.

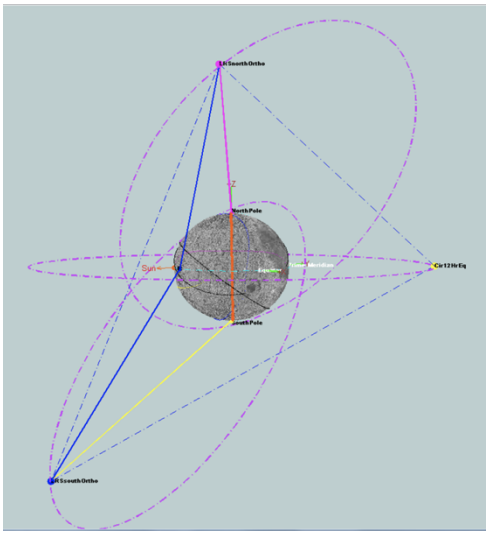

Fig. 5. IOAG Ideal Lunar Relay Satellite Orbits.

Table 1. Ideal Lunar Relay Satellite Orbital Parameters.

Lunar Satellite Orbits	Semi-major axis (km)	Eccentricity	Inclination (deg)	Ascending Node (deg)	Argument of Perilune (deg)	Anomaly (deg)
Equatorial Circular	6124.4	0	0	0	315	Adjustable
Northern Elliptical	6124.4	0.6	57.7	270	270	Adjustable
Southern Elliptical	6124.4	0.6	57.7	0	90	Adjustable

We also made links somewhat limited by frequency bands to create interesting scenarios. For example, in our proxy scenario, we made the Chandrayaan Rover node only able to communicate in S band and the lunar south pole and north pole stations only able to communicate in L and X band. Each node had between 1 – 5 antennas where each antenna could only be used for one channel at a time.

5. BOTTLENECK SCHEDULING

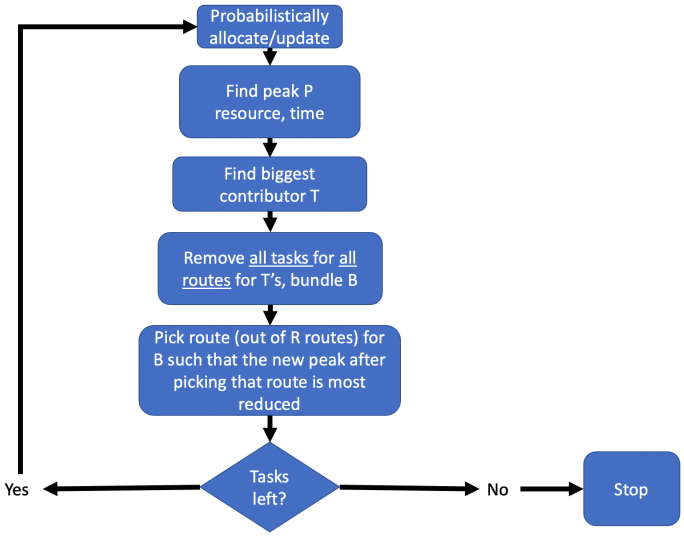

Fig. 6. BNS Flowchart.

Described below is a general Bottleneck Scheduling (BNS) [5] planning algorithm that is applicable to large, complex problems such as satellite communications scheduling, is computationally very efficient, and that we have found performs much better than conventional algorithms, especially when the required tasking strains the applicable resources.

Fig. 6 shows the BNS flowchart. In BNS, first, each task's resource time requirements are spread evenly across all of each possible resource's possible time windows, without regard to when other tasks might be eventually scheduled. These individual profiles are summed across all tasks for each resource. This first step is done to determine which resources are most overburdened and at which times. This is used to determine which tasks should be scheduled first and which time/resource windows should be avoided.

BNS then selects the most overburdened resource, that resource's most overburdened time, that time's task which most contributes to the problem but which has the least flexibility, and finally chooses a resource and time window that minimizes the degree of overburdening most. The intuition is to concentrate on the most contended-for resources at the most contentious times and then to concentrate on the tasks with the least flexibility, since this lack of choice could cause problems later. This might occur if a task with more flexibility were scheduled for a resource at a time where another task absolutely needed it. Finally, the algorithm attempts to make time window choices that minimize the degree of overburdening. In DREAMS's domain, the major resources (satellite sensors) cannot be over-allocated. In some sense, the final plan can have no areas with a height greater than 1, which is why reducing the peaks is important. Although not guaranteed, this algorithm very often generates an optimal schedule in one pass (i.e., other choices could have led to a failure of being able to schedule all tasks), and in rare cases where the schedules are not optimal, they are *very near* optimal.

BNS has been implemented and proven on very complex situations such as scheduling communications between satellites and ground stations [5, 6]. For instance, the algorithm is applicable to continuous and sharable resources, simply by scaling the quantity of resources available from 1 to the appropriate number and scaling the requests for a resource in the same way.

In DREAMS, the BNS resources are transmission and receiving antennas and visibility windows, and the requested tasks are communications tasks that occupy a visibility window for a certain amount of time. The tasks are given by the output of the Route Finder (i.e., the Route Finder outputs R routes, and each route is a sequence of visibility window usages, where each of these visibility usages is a resource). Consider the set of R routes given by the Route

Finder. Within a given route, either all tasks must be scheduled or none must be scheduled. If one route is scheduled (meaning every task for that route is scheduled), all tasks for the other routes for the same bundle must not be scheduled. This affects the BNS in a few ways. First, the initial probabilistic allocation is determined, spreading a bundle's allocation across all tasks for all its routes where each route gets 1/R probability. Second, after finding the peak resource and time and the highest-contributing task T, we must remove all tasks for all routes for the packet associated with task T. Third, when scheduling this task at a resource and time that minimize the new peak, we must consider each of the R routes at a time (i.e., which route, if we added all tasks for the route to minimize the peak after that task is added, yields the lowest new peak).

Note that bundles with tighter deadlines will tend to have fewer routes available, which means each route will have a higher percent allocation during preprocessing, which signals to BNS to focus on inflexible tasks with tighter deadlines first.

6. AURORA SCHEDULING

Aurora handles the fine-grained scheduling. For each route outputted by BNS, Aurora constructs a flow as shown in Fig. 7. A flow is a directed acyclic graph (DAG) representing the start-to-end plan for a bundle. Every flow begins with a Start Flow task. The "main flow" after that consists of repeated sequences of Send, Transit, and Receive tasks, ending with an End Flow task. The "side flow" consists of Store tasks. The Send tasks represent getting the data "onto the channel" – i.e., they do not include the time it takes for the signal to make its way to the receiver. The Transit task represents the time the signal is purely in transit (it has entirely left the sender and none of it has reached the receiver). The Receive task represents receiving the data when it starts to arrive at the receiving node. The Receive and Send tasks will be the same length. The Send task uses the appropriate link resource and a side resource. The Transit task uses no resources (it is only there to model link latency). The Receive task uses a side resource. Note that for some link visibilities, the time spent purely in transit can be zero. If the latency is exactly equal to the length of the Send task, then there will be only a Send->Receive without Transit for that section in the flow. If the link latency is less than the Send task, there will still be Send->Receive, but now they will overlap, with Receive starting partway through the Send.

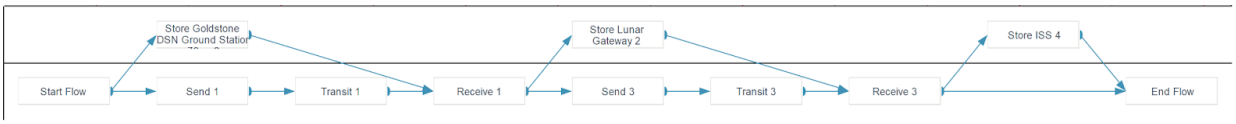

Fig. 7. Aurora DAG Structure.

Aurora is designed as a flexible architecture that can be applied to many different domains and use cases. Therefore, incorporating domain and other knowledge via heuristics is straightforward. For example, Aurora can incorporate prognostics (e.g., antenna X is likely to fail in the next month) when creating/modifying schedules.

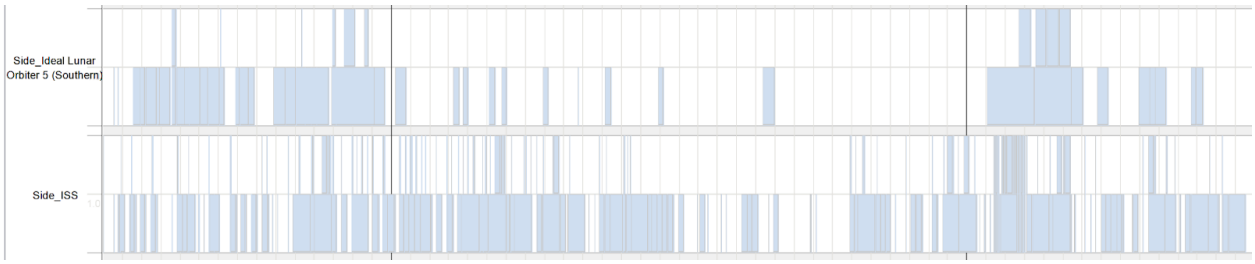

Fig. 8. Example of Aurora Schedule (Sides Grouped By Node).

7. SCHEDULING RESULTS

We show some statistics and plots below to demonstrate the feasibility of our approach.

The DREAMS Router can schedule at varying granularities. As a first step, packets are transformed into bundles. However, for the purposes of scheduling, DREAMS can lump bundles together to form what we'll call "super-bundles." Super bundles really represent a group of adjacent bundles that are routed together. In reality, each bundle within the super bundle will be handled separately in real-time. The amount by which DREAMS aggregates bundles into super-bundles depends on the amount of traffic, the scheduling horizon (how far out to schedule), and the SWaP requirements/capabilities of the node. Note that although super-bundles will be scheduled "in sequence" (i.e., one super-bundle is scheduled for a certain time and another follows immediately after), that does not reflect the routing decision made in real time. The super-bundle may contain many smaller bundles with short deadlines. It may be the case that in real-time, DREAMS will choose to "multiplex" the two super-bundles across both their times in order to meet the individual bundle deadlines. For example, if super-bundle B is scheduled for immediately after super-bundle A, the component bundles of B and A may be interspersed (e.g., two from A then one from B and so on).

For comparison purposes, we also implemented a semi-naïve version the DREAMS Router. The semi-naïve version differs from DREAMS in one way—it considers only the one fastest route for each bundle (i.e., the sequence of links that is capable of providing the fastest delivery time if routing other bundles were not a concern). Note that it still uses Bottleneck Avoidance to pick times within that route (i.e., to pick when during the visibilities on the route to actually transmit the bundle). A truly naïve algorithm would not consider congestion when picking times (instead using priority and deadline only to route bundles). We compared DREAMS's success with the semi-naïve algorithm's success as a way to demonstrate and benchmark DREAMS's capabilities. When compared to a truly naïve router, DREAMS would perform even better, relatively (e.g., in a scenario where DREAMS performs 20% better than the semi-naïve scheduler, it would perform more than 20% better than a fully naïve router).

Example 1

This scenario had 22 nodes: three DSN ground stations; two SN ground stations; three TDRSs (TDRS-K, TDRS-L, & TDRS-M, also known as TDRS 11, TDRS 12, and TDRS 13 respectively); the ISS; a MEO earth satellite; the Lunar Gateway, the five ideal LRSs, the Lunar Communications Pathfinder, the Lunar Flashlight CubeSat, the Chandrayaan Orbiter, the Chandrayaan Rover, a lunar south pole ground station, and a lunar north pole ground station. These formed a total of 300 links with 360 visibilities. We created a schedule that included various types of events representing video calls, audio calls, video data transfer, file transfers (e.g., for software updates), and continuous data streams (e.g., the lunar Gateway's always-on stream of sensor values and other data to earth ground). The amount of generated data varied for different events of the same type (e.g., high-quality video with higher frame rate and resolution per frame vs. lower-quality video, and the same for audio). Events had varying deadlines ranging from seconds to minutes to hours. In addition, some calls were video in one direction and audio in the other (e.g., we had such a "semi-video" call representing a lunar EVA during which video data was sent from Luna to Earth but only audio data was sent from Earth to Luna). We included communication between many pairs of nodes, making the lunar south pole, lunar north pole, Gateway, and the ISS "central" nodes (in that they generated more data and were often bottlenecks for routes for data they did not generate). Earth and lunar ground stations had four antennas, the Chandrayaan Rover had one antenna, the Gateway had five antennas, and all other satellites (earth and lunar) had two antennas. Bit rates were typically about 10 Mbps.

The below scenario over 2 hours runs in 55 seconds (optimizations that we're currently working on will make the runtime multiple orders of magnitude shorter, and these are described later). Plots of the most congested nodes are shown below (i.e., the antennas on the nodes shown below are the busiest in the scenario, due to their originating and/or forwarding lots of data). Each row in each plot represents an antenna on that node. The node name is printed above each plot and the number of antennas is shown below the node name (e.g., "Sides: 2" where we use "Sides" to mean antennas). Each colored bar represents the sending of a super-bundle—the "sending" here refers only to the amount of time required to convert the data into RF, **not** including the latency to get to the receiver nor the time taken by the receiver to receive and process the data. Each "send" task requires use of one side for a certain amount of time. The **semi-naïve algorithm was able to route only 78%** of the bundles DREAMS routed.

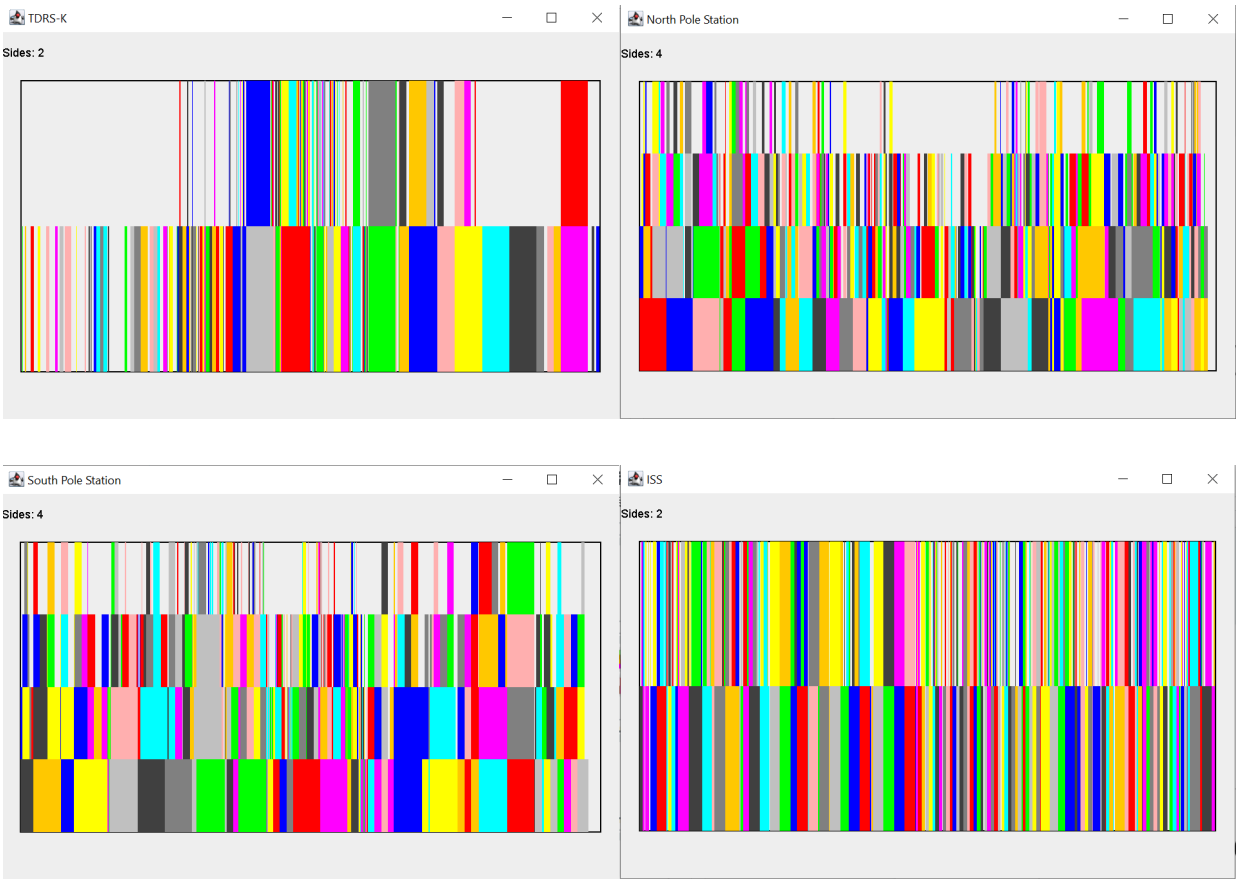

Fig. 9. Selected Resource Activities in Example 1.

Example 2

In this example, we want to demonstrate that our algorithm scales about linearly with the number of nodes. We removed three nodes: TDRS-K (one of the quite congested ones), the Lunar Flashlight, and the Chandrayaan Rover. The plots for the same nodes as above are shown below with the exception of TDRS-K, since it was removed. The runtime on this scenario was 52 seconds, which is quite similar to the 55 seconds in the previous example. This demonstrates that DREAMS is fairly linear. If DREAMS were polynomial, or worse, exponential, the runtime would have dropped more significantly. The semi-naïve algorithm routed only 82% of the bundles DREAMS routed.

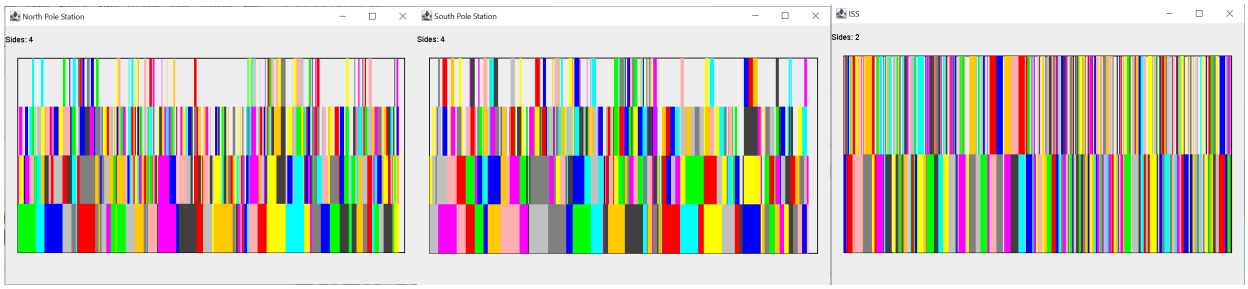

Fig. 10. Selected Resource Activities in Example 2.

Example 3

In this example, we will show examples of the BNS logic. Fig. 11 shows four pairs of bundles (again, really super-bundles, hence the long deadlines). The second always has a much longer deadline than the first.

- Bundle Pair 1 (¼ through the scenario, i.e., 30 minutes in):
  - Metrics for: Short Deadline
    - 3 Gb
    - 30 min deadline
    - **South Pole Station**->Earth-bound Satellite 1
    - Earth-bound Satellite 1->**Canberra DSN Ground Station 70 m**
  - Metrics for: Long Deadline
    - 3 Gb
    - 1 hour deadline
    - **South Pole Station**->TDRS-M
    - TDRS-M->Madrid DSN Ground Station 70 m
    - Madrid DSN Ground Station 70 m->**Canberra DSN Ground Station 70 m**
- Bundle Pair 2 (¼ through the scenario, i.e., 30 minutes in):
  - Metrics for: Short Deadline
    - 10 Mb
    - 10 min deadline
    - **Chandrayaan Rover**->TDRS-L
    - TDRS-L->Lunar Gateway
    - Lunar Gateway->**North Pole Station**
  - Metrics for: Long Deadline
    - 100 Mb
    - 20 min deadline
    - **Chandrayaan Rover**->Ideal Lunar Orbiter 5 (Southern)
    - Ideal Lunar Orbiter 5 (Southern)->Ideal Lunar Orbiter 3 (Northern)
    - Ideal Lunar Orbiter 3 (Northern)->**North Pole Station**
- Bundle Pair 3 (¼ through the scenario, i.e., 30 minutes in):
  - Metrics for: Short Deadline
    - 1 Gb
    - 10 min deadline
    - **South Pole Station**->TDRS-L
    - TDRS-L->Lunar Gateway
    - Lunar Gateway->**North Pole Station**
  - Metrics for: Long Deadline
    - 1 Gb
    - 1 hour deadline
    - **South Pole Station**->Earth-bound Satellite 1
    - Earth-bound Satellite 1->Ideal Lunar Orbiter 2 (Northern)
    - Ideal Lunar Orbiter 2 (Northern)->**North Pole Station**
- Bundle Pair 4 (¼ through the scenario, i.e., 30 minutes in):
  - Metrics for: Short Deadline
    - 1 Gb
    - 15 min deadline
    - **South Pole Station**->TDRS-K
    - TDRS-K->**Goldstone DSN Ground Station 70 m**
  - Metrics for: Long Deadline
    - 1 Gb
    - 30 min deadline
    - **South Pole Station**->TDRS-L
    - TDRS-L->Madrid DSN Ground Station 70 m
    - Madrid DSN Ground Station 70 m->**Goldstone DSN Ground Station 70 m**

Fig. 11. Bundle Pairs for Comparing Chosen Routes.

In general, when a bundle has a longer deadline, it has more routes it can choose from and more times within each link for each route during which it can be scheduled. Therefore, these bundles **tend** not to be high contributors to bottlenecks. This allows BNS to schedule them across low-congestion, longer, further-out routes during non-peak times. As an example, we inspect the routes of the two bundles in Bundle Pair 2. Fig. 12 shows the plots for nodes used for the first bundle's route. Fig. 13 shows the plots for nodes used for the second bundle's route. In both cases, the bundle originates at the Chandrayaan Rover. For the second bundle however, its third node is Ideal Lunar Orbiter 3 instead of the Gateway. The second bundle could also be routed through the Gateway, but since BNS tries to reduce congestion and the Gateway is very congested, it sends it on a different route using the Ideal Lunar Orbiter 3. The second nodes in each path do not differ much in terms of congestion, and it is likely that the Ideal Lunar Orbiter 5 was chosen in the second case as a better way to get to the Ideal Lunar Orbiter 3 (instead of the Gateway).

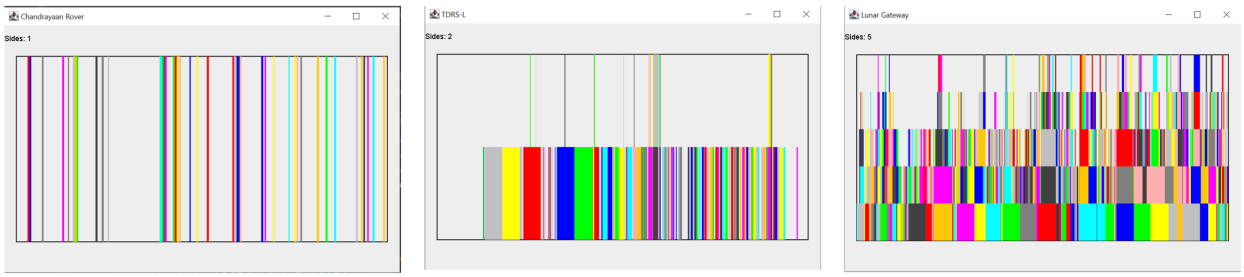

Fig. 12. Plots of Nodes for the First Bundle in Bundle Pair 2.

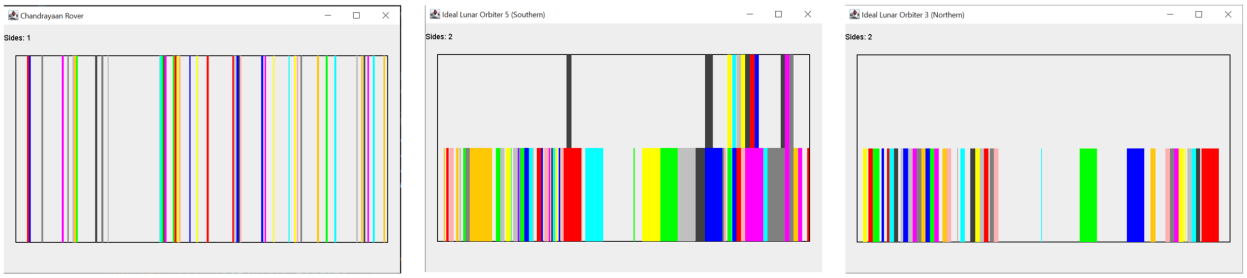

Fig. 13. Plots of Nodes for the Second Bundle in Bundle Pair 2.

Example 4

For this example, we demonstrate that the DREAMS Router handles node downtimes. From the original Example 1 scenario with 22 nodes, we remove one node (TDRS-K) and therefore all links to/from that node. We route about 3500 bundles while the semi-naïve algorithm routes only 85% of what DREAMS could route. The same plots (except TDRS-K since that was removed) are shown below.

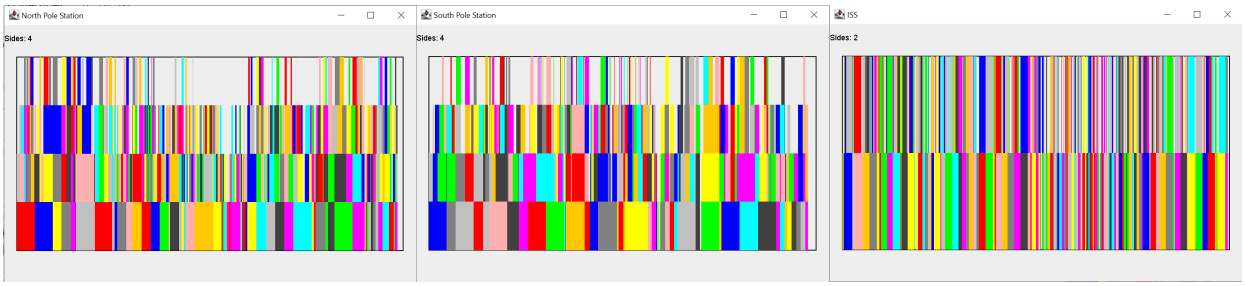

Fig. 14. Selected Resource Activities in Example 4.

Example 5

For this example, we demonstrate that the DREAMS Router handles link downtimes. From the original Example 1 scenario with 22 nodes, we remove one link (we remove both directions between TDRS-K and the ISS). We route about 3600 bundles while the semi-naïve algorithm routes only 84% of what DREAMS could route. The same plots (this time including TDRS-K because TDRS-K still has links with other nodes) are shown below.

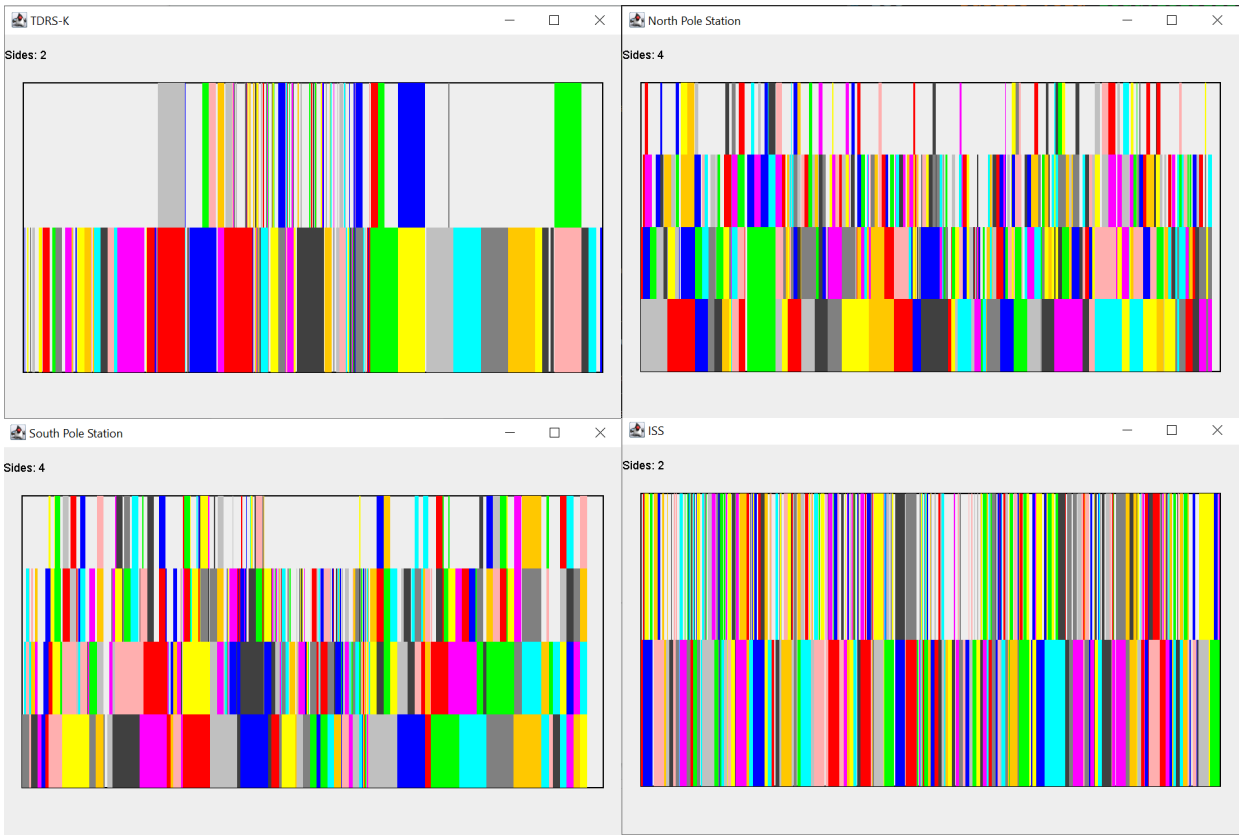

Fig. 15. Selected Resource Activities in Example 5.

10. CONCLUSION

NASA's future missions necessitate maximizing Quality of Service and utilizing multi-hop communications in networks characterized by resource constrained nodes, intermittent links, high latencies, low bandwidths, and ad hoc connections. We presented DREAMS an algorithm that 1) optimizes RF links to increase bandwidths and decrease bit error rates 2) predicts network traffic to help prioritize scheduled high priority transmission 3) a near optimal, distributed scheduling algorithm consisting of two components, Bottleneck Scheduling (an algorithm to select a single route for a bundle) and Aurora (a detailed scheduling algorithm that proposes an executable schedule). This scheduling process grealy increased the total amount of network that could be scheduled over our semi-naïve baseline.

11. References

[1] Taylor, S. J., & Letham, B. (2018). Forecasting at scale. *The American Statistician*, *72*(1), 37-45.
[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.
[3] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.
[4] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269-271.
[5] Mahan, K., Stottler, R., & Jensen, R. (2016). Bottleneck avoidance techniques for automated satellite communication scheduling. In *Infotech@ Aerospace 2011* (p. 1647).
[6] Stottler, R., & Richards, R. (2018, March). Managed intelligent deconfliction and scheduling for satellite communication. In *2018 IEEE Aerospace Conference* (pp. 1-7). IEEE.