Needlecast: On-the-Fly Reconfiguration of Spacecraft Flight Software

Noah T. Curran
University of Michigan, Ann Arbor
ntcurran@umich.edu

Matt Raymond
University of Michigan, Ann Arbor
mattrmd@umich.edu

Wei-Lun Huang
University of Michigan, Ann Arbor
weilunh@umich.edu

***Abstract*—Disruptions in flight software are dangerous, as they can cause a temporary loss of mission-critical functions. In this paper we propose Needlecast, a library for booting NASA core Flight System (cFS) applications on-the-fly, while producing minimal disruptions to software execution. Needlecast results in negligible downtime, maintaining scheduled deadlines for mission-critical processes.**

Our proposed solution has three parts: a sending cFS instance, a network switch, and a receiving cFS instance. Applications in the sending cFS instance regularly broadcast their serialized states to the network switches. Network switches can forward received states (and their associated files) to any registered cFS instance. Receiving cFS instances listen for forwarded data on a global receiving socket, load the given applications dynamically, update their states, and resume their scheduled execution.

Using a single example application, we show that our end-to-end latency is ∼200x faster than loading cFS from a cold restart (the standard alternative). Furthermore, we develop a simple model to determine whether a cold restart or Needlecast is more appropriate, depending on the applications being loaded.

I. Introduction

The NASA core Flight System (cFS) is a widely-used framework for spacecraft flight software. A cFS instance is home to one or more cFS applications, which follow a static schedule to perform periodic tasks while meeting their respective deadlines. Traditionally, cFS applications have been manually installed/updated by operators on the ground. However, this technique leads to significant communication latency and can interrupt a running cFS instance. For significant changes, a costly reboot may be required. There is no current method for storing/transferring states, resulting in significant data-loss on transfer. Mission-critical systems can be temporarily crippled due to this loss of state, leaving the spacecraft vulnerable.

To address this, we must migrate both the application and its execution state from one cFS instance to another. To ensure a smooth transition, both machines should maintain nominal functionality during application transfer, and applications should not be required to recompute data (unless it is specific to the computation environment).

In this paper, we propose a novel method for dynamically migrating application states (application code and its variables) between cFS instances. Our implementation, Needlecast, is an extension for NASA's cFS, and provides a minimalist interface for state migration and dynamic application linking. Using Needlecast, an application can begin running on one cFS instance, transfer to another instance, and continue running with minimal data loss.

This protocol is not an end in itself, but a precondition for many future NASA applications. Once such state-sharing protocols are sufficiently mature, engineers will be able to develop more robust fault tolerance, inter-spacecraft load balancing, frictionless application updates, and continuous integration (CI) practices for spacecraft.

II. Background

A. The NASA core Flight System (cFS)

The core Flight System (cFS) is a reusable framework, code library, and set of programming standards developed at NASA. It was proposed to combat the increasing costs of flight software, and was anticipated to expedite NASA's standard software development cycle. Having a common, highly-modular software framework allows developers to treat programs from past missions as an "app store" (similar to the aptitude repository), reducing development time. Additionally, application reuse eliminates significant testing time, as unmodified applications do not have to be tested again after development.

The cFS framework also provides a method for dynamically linking applications mid-mission. This is incredibly useful, as it empowers developers on the ground to respond to software crises in mid-flight.

B. The NASA cFS Scheduler Application (SCH)

SCH is a specialized cFS application that controls all other applications via a precomputed schedule, and is therefore is included in most cFS instances. This schedule is determined by two tables: the schedule table and the message table. The schedule table describes the static execution order, and specifies what applications will run at what time using major and minor frames. A major frame refers to a single pass over the entire table, and is defined by the Major Time Synchronization signal from the core Flight Executive[1] (cFE) TIME Services. The schedule table also contains a configurable number of minor frames composed of a variable number of software-bus messages. cFS applications create pairs of associated software-bus messages and tasks, and store them in the message table. Applications subscribe to the command pipe, where they listen

[1]cFE is a component of cFS

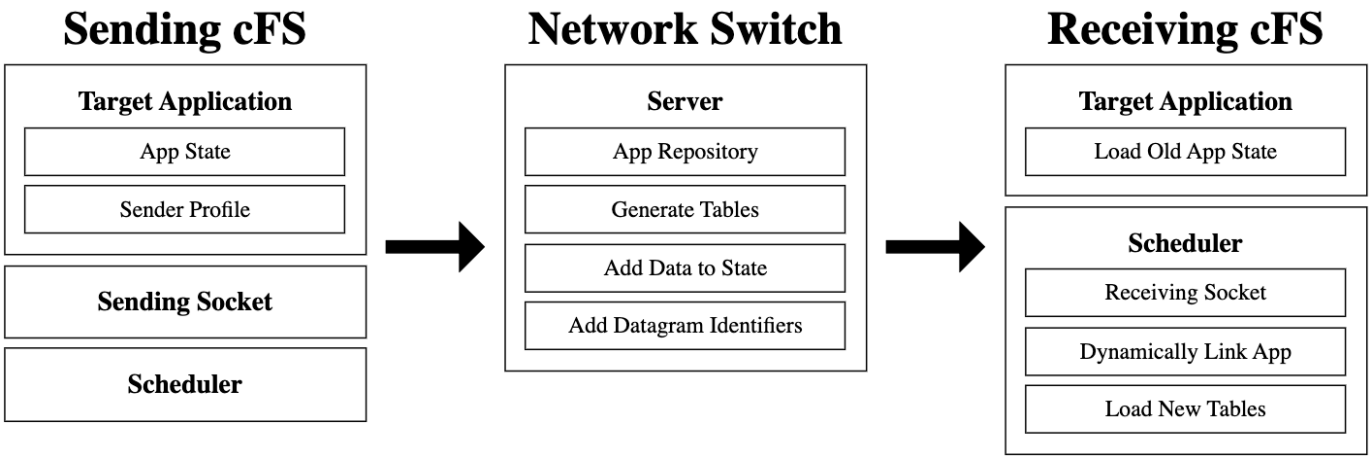

Fig. 1. The system design for our on-the-fly application loading protocol. The left section represents an application that sends its state over the network. The network switch then determines the target machine to send the application and its state to. It also creates and sends a new schedule. The receiving machine then dynamically loads the new application and the new schedule.

for published messages. SCH iterates through the schedule table, and publishes messages to the command pipe. When a cFS application detects an appropriate message from SCH, it begins execution of the associated activity.

By default, the Major Time Synchronization signal operates on a 1Hz frequency. There are 100 Minor Frames in each schedule table, and there are 5 software-bus messages in each minor frame. cFE TIME Services accesses a 1Hz interface and controls the major frame timing. The OS API accesses the hardware timer and controls the minor frame timing.

C. The NASA cFS Applications in General

The design of our cFS application migration protocol is based on cFS application specifications and requirements. Therefore, we will briefly discuss cFS application conventions, and how a cFS instance links an application.

cFS adopts the CMake build system for central management of application tables and header files. To run an application in a cFS instance, the user first pulls the application from its github repository and removes all files except for the source code and `CMakeLists` text file. The user then moves the trimmed application to the `apps` directory in the cFS project. Next, the user modifies the `cpu1_cfe_es_startup.scr` file, specifying the application's main function and shared object (`.so`) file.

Each cFS application acts as an individual program, with a workflow initialized in a `main` function. To simulate event-based listeners, the main function performs a one-time initialization stage, then enters an infinite loop. In this loop, child functions are called to handle software bus messages.

During initialization, cFS links applications as shared object files. This allows ground technicians to perform dynamic linking mid-flight. Our implementation builds on this functionality, allowing us to transfer and link an application without interrupting the execution of cFS.

According to convention, each cFS application stores all of its state data in a single C struct. NASA employees report that all pointer data is instance-specific, so a naive duplication would break the receiving cFS application. Therefore, pointers are ignored during migration.

III. End-to-End Design

Here, we propose an on-the-fly application migration protocol for cFS applications. Although our implementation does not contain all of the necessary infrastructure, we reason about all interfaces in an end-to-end manner. This reasoning shows that the framework is internally consistent and easily extensible in future works. In this context, "sender" is synonymous to "cFS application with state-sending capabilities" and "receiver" is synonymous to "cFS instance with state-receiving capabilities."

A. The Network Switch Assumptions

Our protocol requires at least one network switch, whose implementation is inconsequential as long as it adheres to a few simple rules:

- Each network switch receives UDP datagrams containing application states (as broadcast by cFS applications) and stores them in a database for later use.
- If prompted by some automated/manual processes, the network switch takes a given state from a given cFS application on Host A and broadcasts it to Host B as a series of UDP datagrams. Along with this state, it includes basic metadata for application initialization, a modified pair of a schedule table and a message table, and an the application pre-compiled as a `.so` file for dynamic application linking on Host B.
- Each network switch is assumed to generate the schedule table and the message table as needed, while the cFS-app `.so` file and the metadata are retrieved from a central repository.
- Each datagram sent by a network switch is tagged with an ID. This tag identifies the type of the datagram payload for the receiving end, as shown in Table I and Table II.

TABLE I
STATE DATAGRAM FORMAT

Data	Size
Packet ID (1)	1 byte
Length of App Name	1 byte
App Name (`char []`)	variable
Length of App Entry Point	1 byte
App Entry Point (`char []`)	variable
Stack Size	4 bytes
Priority	2 bytes
State	(variable length, calculated)

TABLE II
FILE DATAGRAM FORMAT

Data	Size
Packet ID (2-4)	1 byte
File	variable (length calculated)

B. The Sender: per application

A cFS instance may host multiple applications. In order to provide individual developers control over their code, we require applications to manage their own states. In our model, each application maintains a state object and its own set of sending parameters. To send its state to the network switches, an application passes the state object and sender profile to the Needlecast sending library.

During initialization, each sender profile uses the sending library to register a sender profile, which includes a unique application ID, a source address, a destination address, and the size of the state in bytes. The sending library links this profile to a new socket and creates an appropriately-sized message buffer, and returns the profile's handle to the application. From here, the message-buffer is managed by individual cFS applications, rather than library code. This alleviates the burden of dynamic memory allocation and generalized code required by a centralized, cFS-wide message buffer.

To perform periodic state broadcasts, the developer's "send state" function must be registered with the message table as a message-activity pair and placed into the schedule table, and the the application must subscribe to the scheduler. When called via message broadcast, the state-sending library will copy the state data to the sender profile's message buffer, adding the application ID and timestamp as metadata.

C. The Receiver: per cFS

On the opposite end, an arbitrary cFS instance acts as a receiving node. It expects incoming data to include a migrating cFS application, the application's latest previous state, and a schedule to accommodate this application.

To properly receive and link application states, the receiver must include the receiving and linking libraries.

a) State Receiving: The receiving library initializes the receiving socket and processes incoming network traffic. For ease of use, we expect this library to be highly modular, with many low-level details abstracted away. Ideally, the cFS application migration functionality should be available through only a few high-level lines of code.

b) Application Linking: The linking library build and updates migrating applications using data passed from the receiving library, resulting in seamless application execution.

The receiving library is initialized upon SCH boot-up, allowing it to listen for incoming states asynchronously. However, receiving library must periodically check the buffer for received datagrams. If a datagram is received, it will be processed by the library and be labeled as one of the following data types:

- The shared object binary of both the migrating application along with application initialization information (e.g., its priority, its stack size, etc.)
- A new pair of schedule table and message table that helps the scheduler accommodate the migrating application's tasks
- The state of the migrated application

Once a full state is received, SCH passes it to the linking library. The linking library stores the incoming state to a global pointer, then links the incoming application. Once the application is linked, the current application state (which may be empty) is replaced with the previously-saved state. Finally, the new schedule and message tables are loaded, and SCH resumes it normal execution with this new data.

IV. END-TO-END IMPLEMENTATION

This section shows our implementation of the aforementioned interfaces. Although Section III presents a complete and general framework, some components (e.g network switches) are out of the scope of this paper. Needlecast, our proof-of-concept implementation, either replaces such components with simplified proxies, or assumes the necessity of such components improbable in the real world.

A. The Network Switch Implementation

We assume that a cFS network switch is automated by highly complex software and thus is out of the scope of this paper. However, Needlecast's testing requires some kind of network switch. Manual emulation using command line tools is too laborious (and requires specialized knowledge), so we develop a web tool for manual control of a simplified network switch. We implement this proto-switch using a NodeJS server on an Ubuntu server, which is assigned a public IP for collaborative purposes. The web interface is built using ReactJS and SocketIO to maintain a constant client-server connection. `lowdb` acts as a local noSQL database for IP addresses and application states. By default, it is hosted on port 8080.

The proto-switch listens on port 8081 for incoming UDP datagrams. Once a datagram arrives, the proto-switch extracts the application ID and timestamp and stores them in the local database. It then displays the application state in a card-view interface (Figure 2) that is updated in real time. The proto-switch only saves the most recent 30 states for simplicity, and

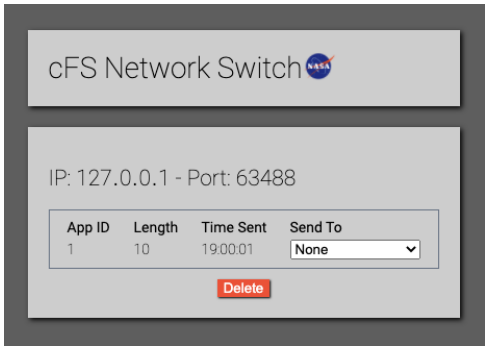

Fig. 2. The Proto-Switch UI

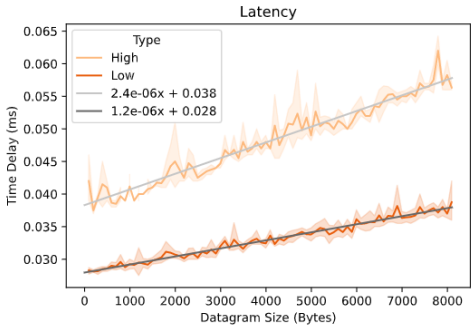

Fig. 3. The latency of the proto-switch in ms as a function of UDP datagram size (Bytes)

old states are overwritten by new states. The interface groups the states by a source IP/port number pair, as each pair should correspond to a single application in a single cFS instance. The interface can also send a state to a specific IP address (corresponding to a given machine) by selecting an option from the ”Send To” dropdown menu. This state, its metadata, and other associated files are then sent to port 8082 at the given IP.

An ideal network switch implementation would include a database for the application state management and distribution. However, such a database is beyond the scope of this paper and also not worth implementing on a temporary proto-switch. Instead, we save files to the disk and hard-code their names into the server code. If a new file is to be sent, it must replace the old file and use the same hard-coded name. Although this limits flexibility, it saves several days worth of development time.

B. Sending the cFS Application States

The state-sending library is implemented via C socket programming, independent of the Operating System Abstraction Layer (OSAL) at the request of NASA employees. The header of each outgoing message contains a 32-bit timestamp and a 16-bit unique application ID. It is unclear whether the application ID should be the result of a hash function, or a centralized application ID naming scheme. For Needlecast, we manually set application IDs.

Since network switches must operate based on packet headers, the state-sending library converts header components to big-endian to respect the networking conventions. We make the simplifying assumption that the endianness of the sending and receiving machines are the same, so the endianness of the payload remains unchanged. Note that if a developer modifies the header component sizes, they should modify the corresponding `hton` and `ntoh` functions as well.

Our minimalist state-sending interface is designed to hide unnecessary details, leaving the application developer with basic variable management. Source/destination address arguments must be strings, and should follow the dotted decimal representation for an IP address and port number. These addresses also extend to the C INADDR_ANY option. After registration of the sending profile, the developer must keep the returned pointer for disposal when the application is ended.

C. Receiving and Handling the cFS Application States

a) The Receiving Socket Library: The receiving library is based on standard C sockets as well, and provides socket initialization/reset functions. By implementing a a non-blocking buffer, polling is almost instantaneous on the average case. When datagrams are found in the buffer, they are extracted into formatted application profiles and stored in a pass-by-reference struct. To prevent memory leaks, previous profiles must be released before new profiles are read from the buffer.

Despite its versatility, the Needlecast makes the following simplifying assumptions:

- No packet drops or duplicates occur
- Datagrams of two application states will not be interleaved
- Each application state, table, and `.so` file fits into one datagram
- There is at most one set of datagrams in the buffer at a time

Each of these assumptions corresponds to a technical problem with well-established solutions, and therefore solving such problems has no research importance. However, we have guarded against two contingencies:

- Datagrams arriving in any order
- Some datagrams arriving much later than others

We consider these cases since they are more likely to happen during testing/benchmarking, and the solutions are trivial.

b) Dynamic Application Linking: In order to load application profiles into cFS, we combine our protocol with the NASA SCH scheduler. Once an incoming state is received, Needlecast sets a global flag that triggers application linking at the next break between minor frames. This results in a slight delay between receiving and linking. An additional delay is caused by the loading of the schedule and message tables.

Our implementation only loads one single application at a time; however, with mutex locks around the global application state pointer, our implementation can sequentially perform dynamic linking to multiple applications.

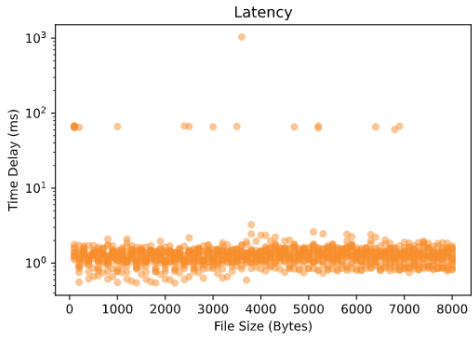

Fig. 4. The latency of the disk write time in ms as a function of buffer size (Bytes). Note that the y-axis is displayed on a logarithmic scale

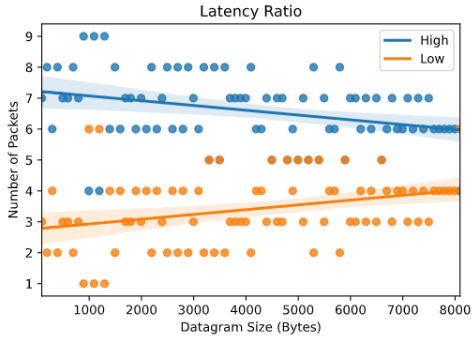

Fig. 5. The number of high-latency delays vs low-latency delays as a function of datagram size

V. Run-time Analysis & Discussion

We now benchmark Needlecast to determine its performance, which we compare to a cold restart of cFS. This benchmark measures the end-to-end latency of various subtasks of the receiver, including:

- Receiving socket to process various datagram sizes
- Saving the `.so` file and tables to disk
- Linking a new application
- Loading new tables into the scheduler

We do not measure the time to send an application's state to the network switches, as this task is completed within is scheduled time frame and does not impact latency at the receiving end. There was also no benchmark performed on the sending network switch, as it is a testing tool and is not part of the official implementation.

All benchmarking is perform on a Raspberry Pi 3B+, which has a Cortex-A53 processor. The Raspberry Pi 3B+ is running the Rasbian operating system with the `PREEMPT_RT` patch installed to emulate a real-time operating system. We utilize the `clock_gettime` function with the `CLOCK_MONOTONIC` parameter to obtain benchmarks of run-time. According to NASA employees, this environment closely resembles what would typically execute flight software.

A. Results

a) Receiving Socket: To benchmark the receiving socket, we programmed the simulated network switch to send multiple packets of increasing size, and record in Figure 3 the time required to load and process the data from the C socket. We did not include time over the network, as this would be indicative of the performance of our network, not our implementation. The data revealed two very distinct trends, which we have split into "High" and "Low" for clarity. These trends are initially split on a rough 2:1 ratio for Low to High, and have distinctly different growth rates. We found that these delays stemmed directly from the unpacking code, but that the exact time that this delay happened seemed to be random and non-deterministic. The long delays appeared to be evenly distributed between all datagrams being unpacked (although the state tended to take slightly longer because of its more complicated formatting). We hypothesized that this could be due to the code being preempted by the OS, but we used the Linux `getrusage` and could find no definite correlation between context-switches and extended delays. However, after regressing the number of High and Low latencies in Figure 5, we see that there appears to be a trend, where the number high-latency packets are increasing. We theorize that this is due to some form of low-level caching that we have no control over.

b) Saving Shared Object and Tables to Disk: To save files to the disk, we use the `fwrite` and the necessary functions for obtaining file pointers. We initially find that disk latency is close to $O(1)$ since writing from the internal buffer to the disk is a non-blocking operation. To ensure proper timings, we use `fflush` and `fsync` to force the file to the disk before continuing. Despite these modifications, timings in Figure 4 are relatively constant as the buffer size increases. Obviously, write time increases linearly with buffer size, but our imput is trivial enough that it gives the illusion of constant time. Note that some write attempts take substantially longer than the average case. We attribute this is to some form of caching, which we have made no effort to control.

c) Loading in an Application: Needlecast is designed to load single applications, however, we are able to simulate multi-application loading by making some slight modifications. Applications are executed in separate processes as they're linked, resulting in parallel execution. A multi-loading version of Needlecast can load applications in series, using a mutex lock to prevent collisions when writing to the global state pointer. As a consequence, the runtime will increase linearly as a function of applications loaded. The average latency for loading a single application is $\sim$1.4 ms, as seen in Table III.

d) Loading in Tables: Before execution, the schedule and message tables must be changed to reflect the newly loaded applications. This operation does not change with the input, meaning that the average runtime is a sufficient benchmark. As seen in Table III, this is $\sim$3.1 ms. This operation need only be performed once, regardless of how many applications are being loaded.

TABLE III
SUB-TASK RUNTIME

Task	Average Runtime (ms)
Load Application	$1.3989n$
Load Tables	3.0858
Save Tables to Disk	6.1623
Save `.so` File to Disk	$3.0811n$
Receive UDP datagrams	$0.024x$ + 0.024 $0.012x$ + 0.012
Booting Up cFS	1315.5718
x = UDP datagram size	n = number of apps loaded

B. Discussion

In this paper, we propose a new method of migrating applications and applications states between cFS instances, and provide Needlecast, an example implementation. This method is intended to replace previous methods that relied on cold restarts and dropped application states. Loading a single application without a mutex lock (since there is no contention for shared resources), we experience a latency of 6.39 ms. This is ∼200x faster than state-of-the-art methods.

However, single application loading is uninteresting. First, there is no mutex lock halting program flow. Second, there may be non-linear scaling, preventing a simple multiplication by single-application latency. Lastly, it's important to understand if there's a point at which is is faster to perform a cold restart.

To calculate this bound, let n be the number of applications loaded in sequence, and x be the cumulative size (in bytes) of all datagrams containing application binaries and tables for all n applications.[2] Given the averages calculated in Table III, our proposed system is preferable when

$$1315.5718 \geq 4.4099n + 0.024x + 9.2721$$

While it is impossible to anticipate the size of `.so` files, if we assume that the binaries are not unusually large, a user could load at least 10 applications before our system becomes slower than a cold restart.

VI. FUTURE WORK

In this paper, we have successfully implemented Needlecast, a method for transferring states and applications between cFS instances via a specialized network switch. To the best of our knowledge, this method is fully-functional. However, there are several areas for improvement.

a) Receiving Socket: Currently, the receiving socket can only process states and files contained within single datagrams. Future versions of Needlecast will allow data to be split amound multiple datagrams and reconstructed upon arrival. Additionally, Needlecast is unable to handle duplicate, dropped, or interleaved packets. This is not a problem for testing, but are considerations to be made before implementing in a real-world setting.

[2]The binaries are separate because they are highly variable in size

b) Other Considerations: Our current state-migration implementation moves only states and applications, and does nothing to affect local files. This was a simplification in our experimental design, but it can potentially cause issues later on. Even if an application needs external data, our state migration will ignore them. In more advanced situations, this can lead to broken applications. Ideally, applications would have an associated list of requirements that would be sent to the receiver. This could either be centrally or locally managed, depending on the mission requirements.

Additionally, Needlecast assumes that it will remain in a close ecosystem where all devices run the same hardware. For example, we assume endianness, word size, and networking capabilities are the same on all devices. This runs contrary to the vision of cFS, and greatly hinders its wider application. In the future, we hope to expand Needlecast to handle all such edge cases cleanly and efficiently.

Currently, applications are agnostic to both their duplication, as well as the time passed since duplication. Temporal-ignorance is fine for most applications, but may be problematic for time-sensitive functionalities. Environmental-ignorance is usually fine as well, but functionalities that require past data to perform current computations (such as a rolling average) could be heavily influenced by changing environments.

VII. CONCLUSION

In this paper we present Needlecast, a cFS library for booting an application on-the-fly. This is developed in an effort to reduce the latency incurred by the current method (cold restarting cFS). This reduction in latency is necessary because of critical flight software services with hard deadlines.

Needlecast consists of three components: an application that sends its state, a network switch that chooses where to send arbitrary state data, and a receiving cFS instance that can dynamically link received applications and states. We use this implementation to validate the usability of our proposed protocol.

During profiling, we determine the runtimes for various components. We find that a single example application has an ∼200x end-to-end latency over current methods. Furthermore, we present an estimated heuristic for determining whether our method is optimal for the given input.

ACKNOWLEDGMENT

We thank Andrew Loveless for his work as our mentor on this project.