ARMY RESEARCH LABORATORY

Real-Time Visualization System for Computational Offloading

by Bryan Dawson and David L Doria

ARL-TN-0655 **January 2015**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TN-0655 **January 2015**

Real-Time Visualization System for Computational Offloading

Bryan Dawson
Oak Ridge Institute for Science and Education

David L Doria
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
January 2015	Final	May 2014

4. TITLE AND SUBTITLE
Real-Time Visualization System for Computational Offloading

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Bryan Dawson and David L Doria

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIH-S
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TN-0655

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
In recent years, the availability and sophistication of mobile computing devices has advanced incredibly. The power of these new devices warrants an exploration of their potential use in tactical environments. Unfortunately, no single device is powerful enough carry out the many computational tasks that will be required on the battlefield. To overcome this limitation, we have started to develop strategies that allow multiple devices to interact and share resources. These strategies include offloading computational tasks to nearby high-performance computers over the tactical network and computation ferrying, or picking up and processing jobs using mobile high-performance computers. These systems and algorithms are in early developmental stages and require complex emulation systems to test their effectiveness. These emulation systems are very useful but do not afford an inherent way to visualize experiments and show the state of the devices and jobs in real time. In this report, we fill this gap by presenting a software package that allows exactly this visualization. In addition, the same package provides the capability to visualize a computational ferrying schedule, which is useful to demonstrate how our scheduling algorithms would be carried out on the battlefield.

15. SUBJECT TERMS
visualization, offloading, network, computation, emulation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Bryan Dawson
Unclassified	Unclassified	Unclassified	UU	22	19b. TELEPHONE NUMBER (*Include area code*) (410) 278-2310

Contents

List of Figures

List of Tables

1. Introduction and Background

With the advancement of mobile computing devices, researchers have begun to test their application in tactical settings and have developed computational offloading strategies to better utilize their computational resources. However, these tests must take place in an emulated environment to effectively control and scale experiments. Our system to emulate offloading scenarios, Computation OFFloading Emulation Environment (COFFEE), tracks and broadcasts its state including the location of devices, network topology, and movement of devices. It also triggers jobs to be offloaded and schedules their computation. All of these factors need to be captured and accurately displayed for developers and other interested parties to understand the decisions the system is making in real time. To do this, we have designed a visualization front end for COFFEE that receives state update commands over a network connection, parses those commands, and translates them into graphical events. The commands are simply strings that indicate the type of event and provide all of the necessary information to visualize the event accurately.

To allow for maximum portability and flexibility for future versions, the visualization system is written in Java 1.7 using standard libraries. This allows for program execution and development on any system that has a Java virtual machine installed. The graphical components use the Java Swing library, which was chosen for its lightweight graphics that are system-independent. Finally, special care is given to maintaining thread safety for all levels of the program, especially the visualization elements, to ensure program responsiveness for the user.

The visualization system is broken into 3 main packages: 1) a communications package that handles all network interaction and stores commands in the buffer, 2) a processing package that handles all parsing of the stored commands and executing the appropriate method calls, and 3) a visualization package that handles all the graphical components. The packages are separated logically so that no dependencies are hard-coded into the visualization system.

The remainder of this report is organized as follows. In Section 2, we discuss the functionality of the background systems, including the network interaction between clients—the JavaServer and the ConnectionHandler, the propagation of commands, and the processing of those commands. In addition, we detail the graphical components used in the visualization system, the logic used to display events graphically, and how the visualization systems handles precomputed demonstrations. In Section 3 we show some sample outputs from our visualization system, and in Section 4 we discuss our conclusions and plans for future work.

2. Software

2.1 Background Services

In our system, background services include everything that occurs between a command being sent from COFFEE and a graphical depiction being displayed on the screen. These services are independent from the user and include all network interaction and the process of turning command strings into graphical events. The following sections detail these services.

2.1.1 Network Communication

The most important functionality of the program is its ability to receive commands from an external application and update the visualization in real time. To accomplish this, the program creates a server that listens on a specific port for clients and binds a socket to the client whenever a client successfully establishes a connection. To allow for multiple, simultaneous connections, the server handles all interactions with clients on separate threads so the server can continuously listen for clients. There is no limit on the number of active clients communicating with the server; the clients only send simple commands that are passed instantly. Once the client is finished sending its commands, the socket is closed and the thread dies.

2.1.2 Propogation of Commands

When clients send commands to the JavaServer, they are read in as strings that must be stored in a shared repository that can handle multiple access requests simultaneously without causing archive corruption. The commands also need to be read into the simulation sequentially to ensure the correct events are triggered in the visualization in the correct order. To accomplish these goals, we use a thread-safe buffer that holds a queue to store the commands. In addition, we created a CommandProcessor that operates on its own thread and continuously requests commands from the buffer. Once a command is received, its format must be validated and interpreted so that the action described by the command can be appropriately visualized. The following table lists all possible commands the system can process.

Table A summary of commands, each presented as a template along with the required parameters to make a successful call, as well as an example of an instance of a command

Command	Type	Param. 1	Param. 2	Param. 3	Param. 4	Example
add;[node];[longitude];[latitude];[isHPC]	Add device	Node	5.97	6.34	False	add;node 1;5.97;6.34;false
remove;[node]	Remove device	Node	NA	NA	NA	remove;node1
link;[node1];[node2]	Link devices	Node1	Node2	NA	NA	link;node1;node2
end_link;[node1];[node2]	End link	Node1	Node2	NA	NA	end_link;node1;node2
start_tran;[node1];[node2]	Start transfer	Node1	Node2	NA	NA	start_tran;node1;node2
end_tran;[node1];[node2]	End transfer	Node 1	Node2	NA	NA	end_tran;node1;node2
add_job;[node1];[job];[active]	Add job to device	Node	Job 1	True	NA	add_job;node1;job1;true
end_job;[node];[job]	End job	Node	Job 1	NA	NA	end_job;node1;job1
remove_job;[node];[job]	Remove job	Node	Job 1	NA	NA	remove_job;node1;job1
move;[node];[longitude];[latitude]	Move device	Node	4.90	7.34	NA	move;node1;4.56;8.22

Note: HPC = high-performance computer

2.1.3 Command Processing

Processing commands requires that all commands conform to a consistent form. When the CommandProcessor reads the command, it is split on a ";" delimiter into a list of substrings. The first element—in this case "add"—is the value that determines the type of event to be visualized. The CommandProcessor then checks that the remaining number of substrings is equal to the number of parameters needed for the method call corresponding to the desired action. If the number of substrings parsed is correct, the method is called in the ApplicationDriver. Otherwise, the command is discarded and the CommandProcessor returns to requesting commands from the buffer.

2.1.4 Visualization Interface

The ApplicationDriver is the last layer between the user and the visualization. When Java creates graphical components, they exist on a single, separate thread. This means that whenever methods are called that affect the graphical components, they must be executed on the Java-designated command thread known as the event dispatch thread (EDT). The ApplicationDriver wraps the function calls in an object that can be run on the EDT so their execution does not disrupt the graphical user interface (GUI) and cause graphical glitches or unresponsive components.

Finally, in Fig. 1, we provide a flow diagram that outlines the full functionality of our visualization system.

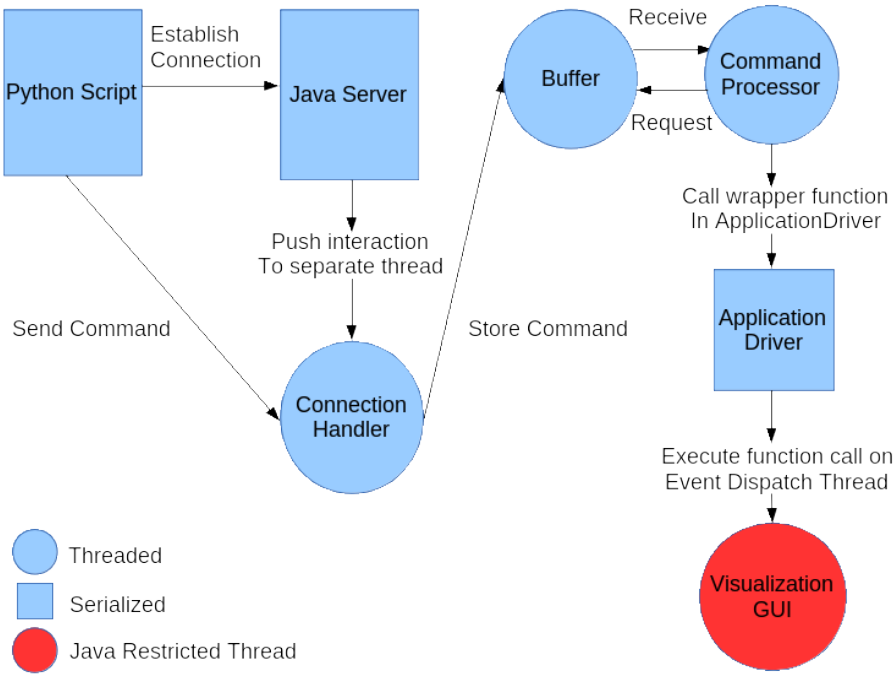

Fig. 1 Flow diagram outlining interaction of all sections of the visualization system

2.2 Visualization Graphics

The visualization components encompass all graphical features that the user can see and interact with while the visualization system is running. They include the following graphical features and will be detailed in the following sections:

- Content Frame: the main program window
- Visualization Panel: the graphical container for all the graphical events
- Individual graphical components: the representation of the graphical events

2.2.1 Content Frame

The GUI components are separated into 2 levels, the topmost of which is the Content Frame, which creates the window space for the visualization and reserves room for an option panel. When emulation scenarios are large, the information displayed in the visualization quickly becomes cluttered and overwhelming. To help the user see only the details relevant to what they are interested in, the options panel provides the following options:

- Show all devices.
- Show only linked devices.
- Show only devices with a job.
- Hide text.

The options panel also contains a historical log of the commands that have been called, whether they were successful, and, if not, why they failed. See Fig. 2 for a detailed view of the options panel showing all the available options and the command field. The most important feature of the Content Frame is that it is independent of the visualization elements and is therefore flexible enough to handle future versions of the visualization software. This is important because it does not require the programmer to rebuild the entire program if new features are added to the visualization and it provides a consistent feel to the user.

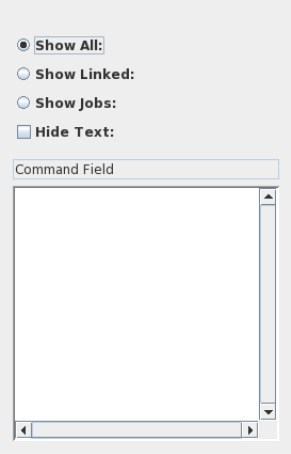

Fig. 2 Options panel

2.2.2 Visualization Panel

The Visualization Panel is the component that handles all of the graphical tasks. The graphical tasks are all defined on devices, which are the graphical objects defined to represents nodes in the offloading emulation. The Visualization Panel provides the functionality for the following:

- Add/remove a device.
- Show/end links between 2 devices.
- Show/end transferring links between 2 devices.
- Show/end/remove pending jobs stored on a device.
- Move a device to a location or to another device.
- Animated movement of a device to a location or to another device.

In Fig. 3, we colorize the interface to show the distinction between the Content Frame and the Visualization Panel. To preserve the GUI thread, efforts are made inside the Visualization Panel to make any function call exit as quickly as possible. For example, constant access data structures are employed, and smart look-up methods (maps, etc.) are used for component access. The Visualization Panel is written so that it is easy to add new features and flexible enough to make use of external graphics libraries in the future.

Fig. 3 Colorized screenshot showing the distinction between the Content Frame (blue) and the Visualization Panel (red)

2.2.3 Visualization Components and Logic

In this section, we outline the logic behind the basic functionality of the Visualization Panel to better illustrate its use as a visual aid for a given simulation.

2.2.3.1 Adding a Device

When a device is added to the simulation, it is defined logically as a node, which is a container for an identification (ID), a longitude and latitude coordinate, and a flag for whether it is a standard mobile device or a high-performance computer (HPC); see Fig. 4 for the graphical distinction. To show the device, first the ID is checked against all other present IDs to determine if that particular device is already present; if it is, the new device is discarded. Once the device has been accepted, its longitude and latitude coordinates are converted to Cartesian coordinates and normalized to the range of values of the devices already present. If the values are outside the current range, every device has its values renormalized—essentially, zooming the view in or out to accommodate the new device. Finally, before the device is drawn to the screen, we check to see if any of the view restrictions are flagged, such as Show Linked or Show Processing. If nothing is flagged, the device is painted to the screen; otherwise, it is cached.

Fig. 4 Graphical representation of a mobile device and an HPC

2.2.3.2 Removing a Device

When a device is removed, the only check required is if the device is present. If it is, it is removed from the list of visible devices and its graphical component is removed. The screen is repainted to immediately reflect the removal of the device.

2.2.3.3 Showing a Network Link

A link is defined as a LinkedNodes object, which is a container for the IDs of the 2 nodes to be linked. This object is created only after the method has verified that the 2 devices are present in the simulation. Then the LinkedNodes object is created and added to a master list of links. During each repaint, we iterate through this master list and draw a line between nodes to represent the links. The iterator also checks that both devices are still present in the simulation. If either device is no longer present, the link is discarded. The benefits of this method of painting a line are that the line will automatically adjust itself whenever either of the devices it is connected to moves and that each device does not have to keep track of all of the devices it is linked to. See Fig. 5 for the graphical depiction of a LinkedNodes object.

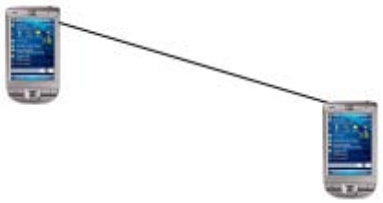

Fig. 5 Graphical representation of a network link

2.2.3.4 Ending a Link

When the command to remove a link is processed, we check to see if both of the devices are present, then check if the link is one of the currently displayed links. If both are true, the link is deleted, each device has its number of active links reduced, and the line will disappear during the subsequent repaint. Finally, we check if the view restriction to only linked devices is flagged and whether each device should be set to invisible.

2.2.3.5 Showing/Ending a Transfer

A transfer can be started only if a link is already established. Therefore, the method checks if both the devices are present and if they are linked. If both are true, the line is updated to be thicker and red to signify that a transfer is taking place. When a transfer is terminated, the same check logic is applied and the line is reverted to its original appearance. See Fig. 6 for the graphical representation of a transfer.

Fig. 6 Graphical representation of a transfer

2.2.3.6 Show/End/Remove a Job

A job is a container for a job ID and a color that represents its status—green for a running job and red for a completed job. If the device associated with processing the job is present, we add the job to the device. The job is represented as a square painted above the device filled with a color representing the state of the job, as shown in Fig. 7. When a job has not yet been processed, it is colored green, and when it is completed, its color is changed to red. In addition, multiple jobs can accumulate on a device and each one can be updated independently to reflect its state. In Fig. 7, we show the different graphical representations of a job on an HPC device. Finally, when a job is removed, its corresponding square is removed from the device.

Fig. 7 Graphical indication of the state of the processing queue on an HPC: a) after picking up a job (job 1), b) after picking up a second job (job 2), c) after job 2 has been processed, and d) after dropping off the completed job 2, job 1 remains in the queue and unprocessed

2.2.3.7 Move a Device to a Location

Moving a device follows the same process as adding a device to the simulation in terms of adjusting locations. If the device is present, it is moved to the new longitude and latitude coordinates and the other devices are adjusted accordingly.

2.2.3.8 Animated Movement

When the visualization system is used to display real-time information, animated movement is not required, as the devices update their positions whenever the system tells them to instantaneously move to a new position. However, when displaying a ferrying schedule, it is

necessary to show the motion between 2 points in a noninstantaneous fashion, actually showing the movement between the 2 points. The animated move method uses a SwingWorker background thread to move the device in a series of small steps across the screen. The number of steps was experimentally set to 50, as this was the fewest number of steps that produced an acceptably smooth motion. A pause is inserted between each step that is 1/50th of the total specified trip time. With this technique, the speed of motion actually varies with the distance travelled. In future work, we will allow the number of steps to vary with the distance so that the speed is constant. Finally, during the animated movement the system prevents the device from being moved by another function call until the device has finished its current movement.

2.3 Precomputed Demonstrations

To run a precomputed (nonlive) demonstration, the user must write a driver that creates events and use Java's timer object to schedule the events to fire at the appropriate time. The tasks are function calls to the ApplicationDriver wrapped in an object that can be executed by the timer. Such a driver has access to all the visualization functionality present in the visualization pane. In Fig. 8, we show a code snippet required to create a single client and a single ferry, instruct the ferry to pick up a job from the client, compute the job, and deliver the result of the job back to the client.

```
public void sample(){

    timer.schedule(new AddDeviceTask("HPC", "12.345", "67.890", true), 0);
    timer.schedule(new AddDeviceTask("NODE 1", "21.543", "76.098", false), 0);

    timer.schedule(new MoveTask("HPC", "NODE 1", 5.0), 3000);
    timer.schedule(new PickUpJobTask("HPC", "job1", true), 8000);
    timer.schedule(new EndJobTask("HPC", "job1"), 9000);
    timer.schedule(new DropOffJobTask("HPC", "job1"), 10000);
    timer.schedule(new PickUpJobTask("NODE 1", "job1", false), 10000);

}
```

Fig. 8 Sample method showing a precomputed ferrying schedule

3. Experiments and Results: Sample Runs

In Fig. 9, we show a sample output from the visualization system constructed from commands.

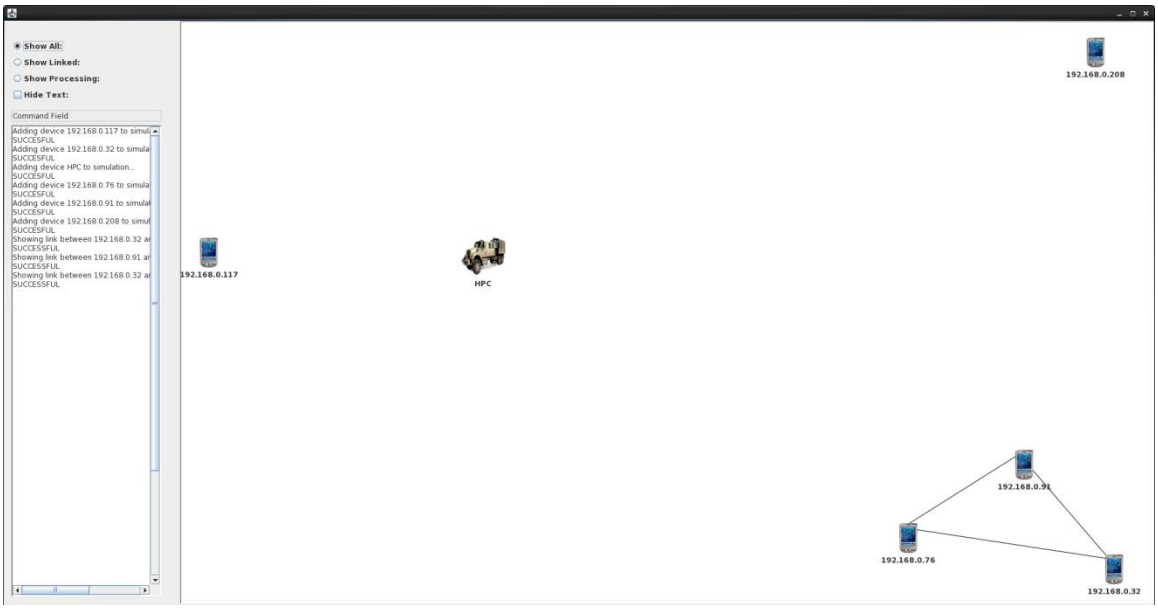

Fig. 9 Sample output of the visualization system. In this scenario, there are 5 mobile devices and one mobile HPC, and there is network connectivity between the 3 devices in the lower right corner.

In Figs. 10–12, we show the state of a precomputed ferrying demonstration during various points.

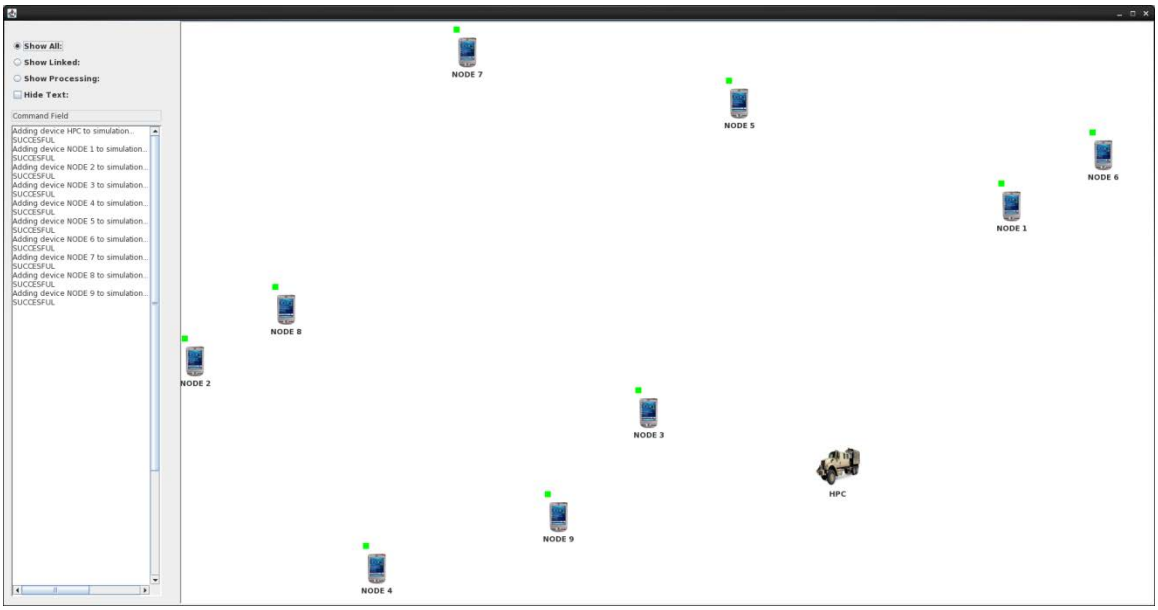

Fig. 10 Sample output of a precomputed computational ferrying demonstration. The demonstration consists of 10 devices—one HPC and 9 mobile devices, each with one active job waiting to be processed.

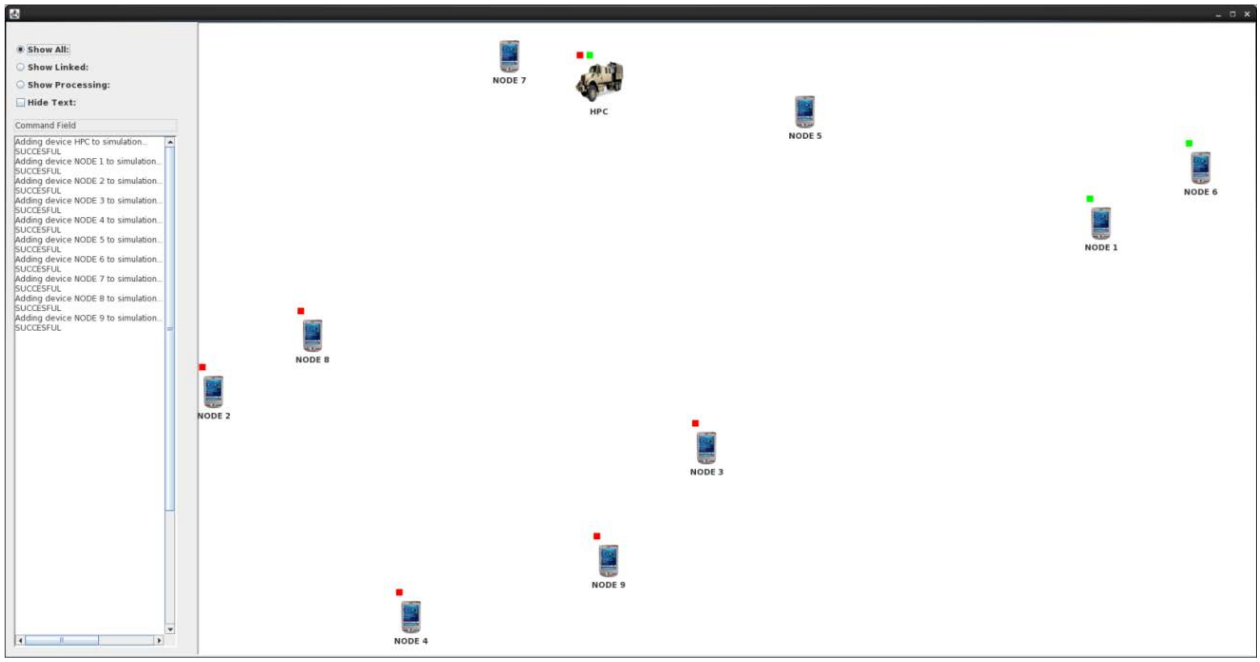

Fig. 11 Sample output of a precomputed computational ferrying demonstration. The HPC has computed and returned jobs for 5 devices and is carrying 2 jobs and finished computing one of them.

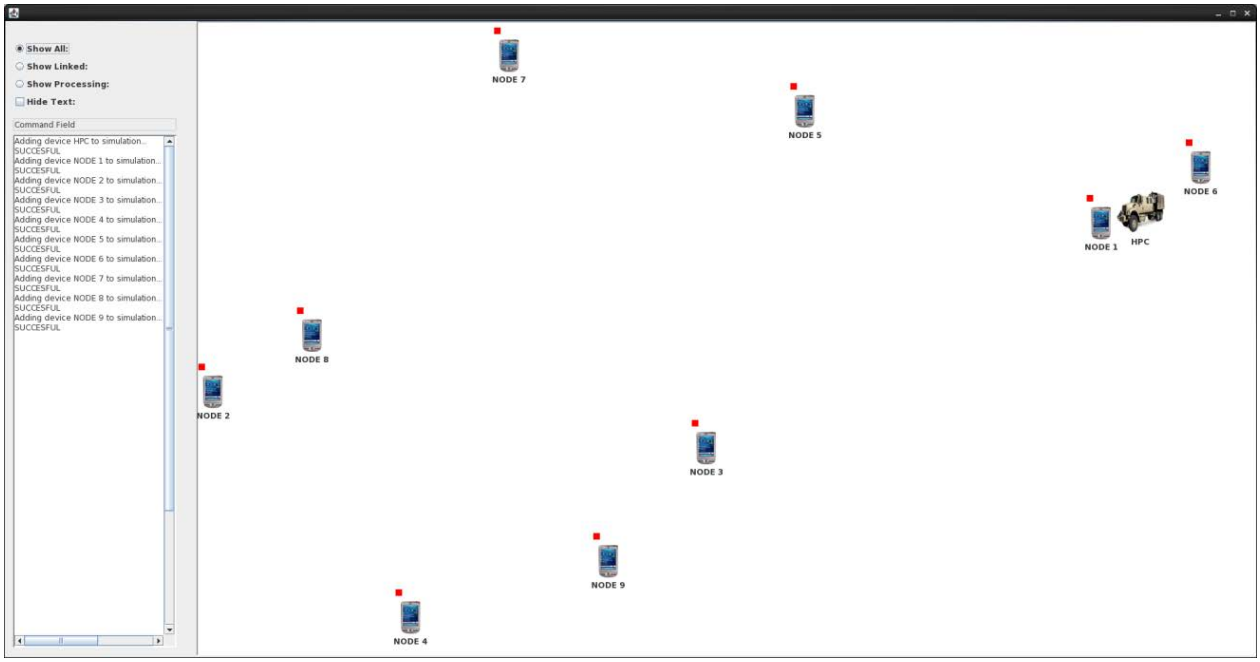

Fig. 12 Sample output of a precomputed computational ferrying demonstration. The HPC has finished computing and dropping off all jobs on the mobile devices.

4. Conclusions and Future Work

We have presented a software system that provides a real-time visualization of computation offloading simulations and computational ferrying schedules. Using command strings sent over a transmission control protocol socket connection, our visualization system can take emulated events from COFFEE and translate them into graphical events. This is accomplished with minimal delay using a thread-safe environment that is portable and lightweight.

Our work will continue by adding graphical features that our current system does not employ so we can better reflect the network traffic and topological information. This includes an animation to show the direction and speed of transfers between devices, adaptive animated movement to account for devices moving inside the simulation, and a custom-cached system to draw images to the screen faster and with less shuttering. We will also integrate our system with the US National Air and Space Administration's World Wind API, a 3-dimensional globe-modeling program, to not only increase the visual appeal of our system, but also to allow us to improve our visualization with terrain data and other realistic information.

1 DEFENSE TECHNICAL
(PDF) INFORMATION CTR
DTIC OCA

2 DIRECTOR
(PDF) US ARMY RESEARCH LAB
RDRL CIO LL
IMAL HRA MAIL & RECORDS MGMT

1 GOVT PRINTG OFC
(PDF) A MALHOTRA

2 DIR USARL
(PDF) RDRL CIH S
B DAWSON
D DORIA

INTENTIONALLY LEFT BLANK.