An Algorithm for In-Situ Bus Scheduling and Passenger State Update

Yung-Fu Chen and Alan Weide

The Ohio State University, Columbus OH 43221, USA
chen.6655@osu.edu, weide.3@osu.edu

1 Introduction

Your final project report should include: 1. A write-up describing

(a) the design rationale of the module(s) you developed, along with the algorithm you have designed/used and a detailed API

(b) instructions and illustration of use, in terms of a README file and/or user guide,

(c) how your implementation has been tested,

(d) performance of your implementation,

(e) assumptions or limitations of your design,

(f) any omissions or errors in your implementation, and

(g) any bugs or weaknesses you have found in the algorithm/protocol. (Only if some of these do not apply to your project should you omit them.)

2. Program sources and executable (including any test programs). Credit will be reserved for the quality of documentation of your programs.

3. Script of a session showing the program under test.

1.1 Problem Specification

1.2 Bus Stop Functions

2 Algorithm

2.1 Request Assignment Algorithm

3 Implementation

This in-situ bus scheduling algorithm was implemented in C# using the .NET Microframework. It makes use of the Samraksh Emulator to simulate a (potentially large) network of buses and bus stops, as well as a sequence of passenger requests.

3.1 .NOW motes vs. Emulator

This project was implemented on the Samraksh Emulator, not .NOW motes. The reasons for this are varied, but one deciding factor was a lack of availability of motes to use in testing and demonstration. Because this project was built as part of the curriculum for a semester-long class, the developers had access to at most three motes. Unfortunately, an interesting demonstration of the algorithm involves at least three bus stops and two buses; that is, five motes would have been the minimum to showcase the capabilities of this algorithm. Thus, the decision was made to implement the project using the Samraksh Emulator, of which arbitrarily many instances can be launched to simulate an arbitrarily large network. Using the emulator instead of motes also enabled some simplifications to the protocol which made it possible to produce a high-quality project in the time allotted.

Simplifications First, by eliminating many distinct pieces of hardware from the equation and consolidating them into instances of an emulator running on a host machine, the system can simplify its timekeeping. Rather than implementing a distibuted time synchronization algorithm, the system can rely on the host's clock to keep time.

Second, by foregoing a physically distributed network of nodes with relatively weak radios and antennae, all messages can be broadcasts (when it is convenient). The emulator's networking capabilities enforce that all messages are broadcast to all other nodes, but in cases where a peer-to-peer communication is desired, the broadcast message is annotated with several pieces of metadata: the kind of message it is, the originating node, and the intended destination node. Instances of broadcasts for the sake of simplicity and convenience are documented in the source code itself, but are detailed here.

One such instance is the response by a Bus when a Bus Stop requests its route information. In a "real" system, due to physical limitations a Bus Stop might only communicate with buses that are nearby; a breadth-first search approach could be used to (eventually) retrieve information about all buses. In our proof-of-concept, each bus will always respond to a broadcast from a stop asking for route information.

Another instance where the broadcast nature of messages is taken advantage of is in the simulation start-up phase. The `RequestDriver` process, upon start up, broadcasts to all nodes in the network the absolute time at which it began sending requests. This is used by each other process as the reference to determine how much simulated time has passed, in order to, for example, determine a bus's current location along its route.

Another simplification enabled by the decision to use the Emulator is that one instance can be reserved as the "passengers". Since each node in the Emulator network can communicate with all the others, a single instance can simulate the requests from many different passengers to many different bus stops. A detailed discussion of the behavior of passengers in our system appears in Section 3.5.

3.2 Software Architecture

Designing the software structure to make use of multiple instances of the Emulator was a focus at the beginning of the project. The structure of the software project is as follows. Each "kind" of process (i.e., Bus, Bus Stop, and Passenger(s)) lives in its own C# *project*, housed within the Smart Bus *solution*. There are 4 total projects in the solution: `Bus`, `Bus−Stop`, `Request−Driver`, and `Utilities And Classes`. The first three, of course, represent the entities involved in the network itself. The fourth project, Utilities And Classes, holds all common components used by the other three projects, including representations of `Bus`, `BusStop`, and `Request`.

Each entity-specific project is composed primarily of a single class, the Driver. The Driver contains an instance of the appropriate entity. The Driver is responsible for sending appropriate messages to the network, and for handling the receipt of messages from the network.

3.3 Message Structure

Due to the broadcast nature of the Emulator network capabilities and a lack of support for peer-to-peer communication, it was necessary to define a relatively complex structure for the messages, which include the originating node, the destination node (if any), the kind of message, and any data associated with the message.

The final project contains a suite of classes relating to messaging in this system. The primary class is `SBMessage`, which defines the overall structure of a message and provides a utility to broadcast the message.

Listing 1. Source code for the `SBMessage` class

```
public class SBMessage
{
    public struct MessageSource
    {
        public enum SourceType
        {
            BROADCAST = 0,
            PASSENGER = 1,
            BUS_STOP = 2,
            BUS = 3
        }
        public SourceType srcType;
        public int srcId;

        // Initializer defaults to BROADCAST with no id
        public MessageSource(string srcType = "0", string srcId =
    "−1")
        {
```

```
            this.srcType = (SourceType)int.Parse(srcType);
            this.srcId = int.Parse(srcId);
        }

        public MessageSource(SourceType srcType = SourceType.
    BROADCAST, int srcId = -1)
        {
            this.srcType = srcType;
            this.srcId = srcId;
        }

        public override string ToString()
        {
            return this.srcType.ToString() + "␣" + this.srcId.
    ToString();
        }
    }

    public struct MessageHeader
    {
        public const int Length = 5;

        public MessageType type;
        public MessageSource origin;
        public MessageSource destination;

        public MessageHeader(MessageType type, MessageSource
    origin, MessageSource destination)
        {
            this.type = type;
            this.origin = origin;
            this.destination = destination;
        }

        public override string ToString()
        {
            return type.ToString() + "␣" + origin.ToString() + "␣"
     + destination.ToString();
        }
    }

    public enum MessageType
    {
        START_SIMULATION = 10,
        SEND_PASSENGER_REQUEST = 11,
```

```
    ROUTE_INFO_REQUEST = 20,
    ROUTE_INFO_RELAY_REQUEST = 21,
    ROUTE_CHANGE_REQUEST = 22,
    REQUEST_SCHEDULED = 23,

    ROUTE_INFO_RESPONSE = 30,
    ROUTE_CHANGE_ACK = 31
}

public MessageHeader header;
public IMessagePayload payload;

public SBMessage() { }

public SBMessage(MessageType type, MessageSource origin,
MessageSource destination, IMessagePayload payload)
{
    this.header = new MessageHeader(type, origin, destination)
;
    this.payload = payload;
}

public SBMessage(string msgString)
{
    string[] components = msgString.Split();
    MessageType msgType = (MessageType)int.Parse(components
[1]);
    MessageSource source = new MessageSource(components[1],
components[2]);
    MessageSource destination = new MessageSource(components
[3], components[4]);
    int headLength = MessageHeader.Length;
    switch (msgType)
    {
            // Message from Passenger
        case MessageType.START_SIMULATION:
            this.payload = new PayloadDateTime(components, ref
 headLength);
            break;
        case MessageType.SEND_PASSENGER_REQUEST:
            this.payload = new Request(components, ref
headLength);
            break;
```

```
                // Message from Bus Stop
            case MessageType.ROUTE_INFO_REQUEST:
                this.payload = new PayloadSimpleString();
                break;
            case MessageType.ROUTE_INFO_RELAY_REQUEST:
                this.payload = new PayloadRouteRequestForward(
    components, ref headLength);
                break;
            case MessageType.ROUTE_CHANGE_REQUEST:
                this.payload = new Route(components, ref
    headLength);
                break;
            case MessageType.REQUEST_SCHEDULED:
                this.payload = new Request(components, ref
    headLength);
                break;

                // Message from Bus
            case MessageType.ROUTE_INFO_RESPONSE:
                this.payload = new Route(components, ref
    headLength);
                break;
            case MessageType.ROUTE_CHANGE_ACK:
                this.payload = new PayloadRouteChangeAckResponse(
    components, ref headLength);
                break;
        }
    }

    public override string ToString()
    {
        StringBuilder msg = new StringBuilder();
        msg.Append(this.header.ToString() + "␣");
        msg.Append(this.payload.BuildPayload());
        return msg.ToString();
    }

    public void Broadcast(NetInst NetPort)
    {
        Debug.Print("Sending␣message␣" + this.ToString());
        byte[] msgBytes = Utilities.StringToByteArray(this.
    ToString());
        NetPort.Broadcast(msgBytes, msgBytes.Length);
    }
}
```

Also required was an interface, `IMessagePayload`, to be implemented by any component which might need to be included in a message in this system. Such components include `Route`, `DateTime`,[1] `Request`, and others.

3.4 Testing and Evaluation

3.5 Limitations, Challenges, and Omissions

Passenger Behavior Initially, the plan for simulating passengers was to randomly generate requests on the fly, which would be served and assigned by appropriate bus stops. However, the inherent limitations of the .NET Microframework proved insurmountable given the relatively short time frame.

In lieu of dynamic passenger requests, a `RequestPattern` interface was devised which permits the arbitrary, static instantiation of passenger requests, to be processed in real-time by the `RequestDriver` Emulator instance. For each request in the pattern, the driver schedules a timer to fire at the time when that passenger is intended to dispatch the request. However, the `RequestPattern` interface is general enough that an implementation could, theoretically, generate random requests on the fly, though that implementation does not appear in this project.

3.6 Using the Smart-Bus Application

4 Conclusion

[1] A payload with type `DateTime` is actually a wrapper type, `PayloadDateTime` due to C# restrictions on extending `struct`s.

A Source Code