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 Assignment Description

Your final project report should include:

- A write-up describing
  - The design rationale of the module(s) you developed, along with the algorithm you have designed/used and a detailed API
  - Instructions and illustration of use, in terms of a README file and/or user guide,
  - How your implementation has been tested,
  - Performance of your implementation,
  - Assumptions or limitations of your design,
  - Any omissions or errors in your implementation, and
  - 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.)
- Program sources and executable (including any test programs). Credit will be reserved for the quality of documentation of your programs.
- Script of a session showing the program under test.

2 Introduction

Internet of Things (IoT) has been an innovative technology connecting heterogeneous devices to form a practical network for facilitation of human work. In IoT, mobile adhoc network (MANET) is a key point to give the greater mobility for users and reduce deployment costs of the network. Typically, Wi-Fi and IEEE 802.15.4 are the popular communication technologies for different application requirements. Wi-Fi technique provides good throughput in wireless communication but limited transmission range. On the other hand, although IEEE 802.15.4 supports low data rate with available range of dozens of meters, it has the property of low cost and low power consumption. However, there are applications requiring data traffic routing in a wide area of a few kilometers. For example, in fleeting tracking system, each car is regarded as a mobile node in a MANET. The average hop distance between two adjacent nodes could be in kilometers. Thus, it is infeasible to adopt Wi-Fi or IEEE 802.15.4 techniques to guarantee the connectivity of network in such application. Thus, several transmission technologies have been introduced to satisfy the requirement of long transmission range. Low-Power Wide-Area Network (LPWAN) is a type of wireless telecommunication wide area network designed to allow long range communications at a low bit rate among connected objects with battery power. Like Long-Range (LoRa) technology, it is a LPWAN specification intended for wireless battery-operated objects and suitable for these applications.

Due to the maturity and popularization of Wi-Fi in mobile devices rather than the technology of LPWAN, we proposed an idea to take advantage of both Wi-Fi and LPWAN and combine them into an access point in MANET, so that mobile devices, such as cellphones and tablets, can directly use Wi-Fi to access the network via this kind of access point. Also, because of the property of LPWAN, the hop distance between neighbor access points can be long enough to reduce the hop count in routing. Moreover, we will introduce an innovative application, named Smart Bus System to show how the whole network composed by this kind of access points can support low data traffic, energy efficiency, dynamic scheduling, and efficient routing. Also, this paper aims to propose a distributed algorithm which efficiently solves the bus scheduling problem in the smart bus system so that the total travel time, sum of lateness over all pick-up and delivery locations, and sum of overtime over all buses can be minimized.

2.1 Smart Bus System

Compared to typical bus system with fixed route, a smart bus system is required for dynamically calculating the route to pick up passengers and efficiently shorten the regular route to save more waiting time of passengers as well as energy consumption of buses. We proposed low power Bus Service Access Points (BSAPs) with Wi-Fi and LPWAN to establish a mobile ad-hoc network with the ability to provide high speed transmission in short distance below 30 meters and low speed transmission in long distance in kilometers.

In this system, each bus and bus stop are equipped with one BSAP. A BSAP can offer limited Wi-Fi service for pick-up service and browser-based application for user to send a request or query service information. A passenger with cell phone can directly use Wi-Fi to access the BSAP to send a request for pick-up service at any bus stop on campus. The request includes the passengers origin stop, destination stop, the earliest departure time from origin and the latest arrival time to destination. After receiving a request, the serving bus stop needs to determine the matched bus according to the pair of pick-up and delivery stops and its time constraints. On the other hand, after receiving a request, a BSAP on bus has to determine whether to serve the passenger or not. If the bus accepts this request, it will recalculate a new route to pick up a set of passengers and satisfy their expected arrival time as possible. Otherwise, if the bus rejects the request due to it limited capacity, the corresponding serving bus stop has to reassign the request to other available buses for seeking the best utilization.

Without attatching an additional LPWAN interface, a passenger will always use Wi-Fi radio to communicate with a BSAP on bus or bus stop. On the other side, a BSAP can exploit the Wi-Fi interface with higher throughput to transmit message if the distance between two neighbor BSAPs is within an available Wi-Fi range. Otherwise, a BSAP can always use LPWAN radio to communicate with any neighbor BSAPs due to its long transmission range in kilometers. We assume all BSAPs on bus stops form a static connected network. Thus, the mobile BSAPs on any buses moving within the area of static connected network can still be under the network coverage so that the matched bus can always received the request from bus stops regardless of its location.

3 Problem Specification

In this bus system, a passenger will send the adjacent bus stop a request including his origin stop, destination stop, the earliest departure time from origin and the latest arrival time to destination. Hence, to implement the system and meet the requirement from passengers, we need to solve the scheduling problem stated in the following sections.

3.1 Scheduling Problem

Let p_i^+ and p_i^- respectively denote the pickup point and the delivery point of passenger i. Also, we have earliest departure time ED_i from point p_i^+, latest arrival time LA_i to point p_i^-. The problem consists of a set $B = \{1, 2, \ldots, m\}$ of m buses in the smart bus network with their current routes. For the route of bus k, it includes a combination of multiple pickup points and deliver points, which is denoted by $R_k = \{p_0, p_1, \ldots, p_r\}$, where $p_i^+ \in R_k$ if passenger i is matched to bus k and he has not been picked up and $p_i^- \in R_j$ if passenger i is matched to bus k and he has not reached his destination. The order of points in R_k depends on the earliest departure time and the latest arrival time of matched passengers to bus k. Suppose p is the number of matched passengers to bus k who have not been picked up and q is the number of matched passengers to bus k who have not reached their destinations. Then $p \geq q$ because there might exist some passengers who have been picked up but not reached their destinations (but there are no passengers which have reached their destinations that have not yet been picked up). For a bus k, it has limited capacity to serve passengers. Assume all buses in the network have the same capacity. Let $c_{k,v}$ be the number of seats available in bus k when arriving location v. $c_{k,v}$ decreases by 1 after serving location v which is a pickup location. Otherwise, $c_{k,v}$ increase by 1 after serving a delivery location. A bus k cannot serve any passengers at a pickup location v if $c_{k,v} = 0$.

Moreover, let $t_{a,b}$ be the direct travel time between points a and b. For each service location v (either a pickup or delivery point), we compute time windows of this service location $[e_v, l_v]$, which denotes the earliest and latest served time. For example, for a request i with pickup point p_i^+ and delivery point p_i^-, we can calculate the time window of p_i^+, $[ED_i, LA_i t_{p_i^+, p_i^-}]$ and the time window of c_i^-, $[ED_i + t_{p_i^+, p_i^-}, LA_i]$. In the system, each bus starts and ends its route at the *bus terminus* (location 0). Let ts_k and te_k denote start time and end time for bus

k. There is also a time window at the terminus, where e_0 and l_0 are the earliest start time and latest end time of each route.

Let t_v denote the serving time at point v by the matched bus. It denotes the exact pick or delivery times for passengers. If v is the predecessor of v in a route, the arrival time of the bus k at v is $t_v = max(t_v, e_v) + t_{v,v}$; otherwise $t_v = ts_k + t_{0,v}$.

Since the requests from passengers are dynamic and independent, the problem is to find a matched bus for a newly added passenger in real time while the system receives a request from him and to adjust the route of the matched bus in order to minimize the total travel time of buses, the sum of "lateness" over all pick-up and delivery locations, and the sum of overtime over all buses. Thus, we can define the problem as the statement provided below.

Given a set $B = \{1, 2, \ldots, m\}$ of m buses on the route. Each bus k has the current route $R_k = \{p_0, p_1, \ldots\}$. Whenever receiving a request from passenger i with the parameters p_i^+, p_i^-, ED_i, and LA_i, we compute the matched bus and insert the pickup point p_i^+ and delivery point p_i^- to the route R_j of matched bus j according to the time windows of service location $[e_v, l_v]$. The goal is to minimize the objective function:

$$min\{\sum_{k\in B} T_k + \alpha \sum_{v\in V} max\{0, t_v - l_v\} + \beta \sum_{k\in B} max\{0, te_k - l_0\}\} \tag{1}$$

T_k denotes the total travel time of bus k. The weights α and β define the relative importance of the different components.

4 Algorithm

4.1 Route Flexibility

Since the goal is to minimize the objective function, whenever assigning requests from passenger, a serving bus stop needs to compare a metric of buses to select the best bus leading to the minimized result of objective function. Thus, we define a metric, *route flexibility*, to evaluate the flexibility to accommodate a new request into the current route. Let $f_{k,v}$ denote the flexible time of bus k at location v, which is defined by $l_v t_v$. That is to say, if the arrival time t_v is much smaller than the latest served time l_v, then the bus has the higher flexibility to accept more requests. Hence, to calculate the route flexibility, we use F_k to denote the accumulated flexible time of service locations in R_k.

$$F_k = \sum_{v\in R_k} f_v + (l_0 - te_k) \tag{2}$$

$l_0 - te_k$, the flexible time at terminus, decreases or stays the same whenever a new request is added to R_k. Note that $\sum_{k\in B} F_k$ (i.e., the total flexibility) is inversely proportional to objective function. Hence, objective function can be minimized if the total route flexibility is maximized during request assigning for each bus stop.

4.2 Route Information Sharing

When assigning requests, the serving bus stop must know the current route information of buses and their current capacity. For example, if bus k picked up a passenger at location v, then the number of service location in R_k decreased and the capacity c_k decreased as well. At that time, bus k should update the information to bus stops.

4.3 Request Assignment Algorithm

With the route information of buses, a bus stop is able to compare the route flexibility of different buses after inserting a request to their routes. However, when receiving a new request, instead of immediately assigning it to a bus, the serving bus stop will wait for a certain amount of time to determine which request should be served first and which bus is the best one to serve the request. This is because all requests are dynamically launched by passengers, bus stops do not know requests in advance until receiving them. To maximize the total flexibility as possible, bus stops should serve requests according to the urgency rather than the receiving order since we know an urgent request can harm the route flexibility if it is not immediately assigned to a bus. On the other hand, the request, with larger latest arrival time, can be reasonably delayed for a certain time because buses may need to reserve space for a potentially urgent future request.

Let δ denote the urgency threshold. For a request i, if $LA_i - \delta$ is smaller than the current time, we can say it is an urgent request. For assigning urgent requests, a serving bus stop assign the request to the bus with maximal route flexibility after inserting the request. All other requests will not be assigned until all urgent requests have been dispatched to the matched buses. Furthermore, for non-urgent requests, if there exists a request with p_i^+ and p_i^- matching the service locations in current route R_k of bus k, the serving bus stop will prioritize the request to be served first. The flexible time at terminus ($l_0 te_k$) remains the same after inserting this request. Thus, the flexibility will not be harmed.

5 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.

5.1 .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 one bus. Evaluating the efficacy and scalability of the algorithm with such a limited number of nodes would be impossible. 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 5.4.

5.2 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.

5.3 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.

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.

5.4 Limitations, Challenges, and Omissions

Network Topology In an effort to dramatically simplify many of the calculations involved in this algorithm, this version of the system assumes that all stops are arranged in a line, and that buses can travel directly only between neighboring stops. The algorithm as specified, of course, can handle arbitrary network topology, but our simplified implementation cannot. However, there are artifacts in the source code which were placed with the express purpose of ensuring this system remains extensible, and so could be adapted to handle arbitarily complex network topologies.

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

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

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.

5.5 Testing and Evaluation

The efficacy of this (admittedly complicated) distributed algorithm versus a naive, static scheduling approach was evaluated. The static scheduling approach is similar to that employed by real public bus systems in the world: there are N buses, each with a predetermined route R through a number of stops.

Evaluation metrics include

- The number of buses required to service k passenger requests on time (lower is better)
- The number of requests that can be served on time with n buses (higher is better)

Request Patterns A variety of request patterns were used for evaluation. The scale of these patterns range from two passengers to ???, from two bus stops to ???, and from two buses to ???.

Specification and Evaluation of Static Approach The control for this experiment was a static, "dummy" approach (which we call the *Traversal* approach) in which N buses traverse the line of stops from one end to the other, and then back again. Its efficacy was evaluated from a purely theoretical standpoint; there is no implementation of the static algorithm. For each Request Pattern on which the Smart-Bus algorithm was tested, the minimum number of buses required by the Traversal approach to serve those requests on time (or, with minimal lateness) was computed. The results are summarized in Table 1 below.

The method for computing the values in the table is quite simple, given the assumptions made about the toplogy and travel time between stops. For a traversing bus B_i, its route $R = \langle 0, 1, 2, \ldots, N-1, N, N-1, N-2, \ldots, 1, 0, 1, 2, \ldots \rangle$, repeated until the total travel time is greater than its operating window: $T_{B_i} > (l_0 - e_0)$. Thus, given a system with hop duration H and stop duration S, the number of stops along a bus's route is $B_S = \frac{l_0 - e_0}{H+S} + 1$ and the number of traversals made by a bus is $B_T = \frac{B_S}{N}$, and the time it takes for a traversal in a system with N stops is $T_N = N(H+S) + S$.

Since a bus will never change its route, it is trivial to predict exactly where it will be along its route at a given point in time. This information can then be used to determine, at the time a request is made, when the next bus to that stop will arrive and how long it will take to arrive at the request's destination. If any request cannot be served in time, an extra bus must be placed in the network.

By repeatedly applying these calculations, we compute the minimum number of buses required to serve all requests in time.

Table 1. Evaluating the Traversal Approach.

Request Pattern	Min # of Buses	Total Lateness
2P_2B_2S	?	?
4P_2B_2S	?	?
6P_4B_4S	?	?
8P_4B_8S	?	?

6 Using the Smart-Bus Application

There are several steps to running the application:

– Launch n instances of `Bus−Driver`
– Launch m instances of `Bus−Stop−Driver`
– Ensure the simulation parameters of `Request−Driver` match the number of buses and stops that have been launched
– Launch an instance of `Request−Driver` with the requisite simulation parameters (i.e., Request Pattern and Time Multiplier)
– Wait for simulation to conclude

6.1 Extending the Smart-Bus Application

It is, of course, possible to extend or improve the functionality of this application in many ways.

– For work on generating dynamic requests, implement the `RequestPattern` interface
– To change the request assignment behavior, modify the `BusStop` class in `Utilities and Classes`
– To implement breadth-first search to find buses, modify the `BusStopDriver` class
  • Changes to `Bus` will also be necessary to restrict its responses to those stops within its immediate vicinity

7 Conclusion

A Source Code