Measurements of a Real-time Transit Feed Service Architecture for Mobile Participatory Sensing

Róbert Szabó*†, Károly Farkas*‡ and Bernát Wiandt‡
*Inter-University Centre for Telecommunications and Informatics, Debrecen, Hungary
†HSNLab, Dept. of Telecommunications and Media Informatics,
‡Dept. of Networked Systems and Services,
Budapest University of Technology and Economics, Budapest, Hungary
Email: szabo.robert@etik.hu, farkask@hit.bme.hu, bwiandt@hit.bme.hu

***Abstract*—We spend a substantial part of our time with traveling, in crowded cities usually taking public transportation. It is important, making travel planning easier, to have accurate information about vehicle arrival times at the stops. Most of the public transport operators make their timetables freely available either on the web or in some special format, like GTFS (General Transit Feed Specification). However, they contain static information only, not reflecting the actual traffic conditions. Mobile participatory sensing can help extend the basic service with real-time updates letting the crowd collect the required data. With this respect we believe that such participatory sensing based application must offer a day zero service following incremental service extension. In this paper, we discuss how to realize real-time refinements to static GTFS data based on mobile participatory sensing. We show how this service can be implemented by an XMPP (Extensible Messaging and Presence Protocol) based mobile participatory sensing architecture and we evaluate its performance.**

***Keywords*—Participatory sensing, Public transport, GTFS, Publish/subscribe, XMPP**

I. Introduction

Traveling is an everyday activity for most of us. Using public transportation is an environmentally sound way of traveling. To reduce unnecessary waiting times accurate estimates of vehicle arrivals are to be provided to the passengers. Fortunately, most transit operators of big cities make their timetables freely available in the form of General Transit Feed Specification (GTFS) [1]. However, this solution contains only static information, which does not reflect up-to-date traffic conditions.

Exploiting the emerging paradigm of mobile crowdsensing [2], or often called participatory sensing, the crowd of people collects the necessary data, such as delays or traffic jams, to make live updates available for real-time community services. For sensing, the built-in and ubiquitous sensors of the passengers' mobile phones can be used. Every traveler can contribute useable information. Thus, travelers standing at a stop can send data with regard to each arriving and departing bus/tram/train. While passengers in travel are expected to collect periodic position and stop arrival/departure information relevant to a given vehicle only.

However, using crowdsensing faces substantial challenges. The primary challenge is the motivation of travelers to be involved in gathering the necessary information. Our approach is a day zero service, which means some basic functions offered to its users from the day the service is launched, and improved functions are added as data is gathered by the crowd. We believe, that this approach may provide appropriate incentives to the users.

Thus, in this paper, as the follow-up of our former work [3], [4], we investigate and discuss an application scenario, where the day zero service is a static transit feed which is incrementally improved with live updates as data is collected from passengers. We present the implementation of this scenario using an Extensible Messaging and Presence Protocol (XMPP) [5] based service architecture. Moreover, we investigate the performance of our service setup. Our measurement results show that even two-three year old commodity hardware is able to cope with the generated traffic load. With a GTFS emulator and the proposed setup we can combine static and real-time service updates in an easy way and introduce incremental service improvements by the aid of mobile crowdsensing.

The rest of the paper is structured as follows. In Sec. II we review related work. We introduce our design in Sec. III. We present some preliminary measurement results in Sec. IV. Finally, we summarize the paper in Sec. V.

II. Related Work

In this section, we review first crowdsensing based public transport tracking applications. Then we discuss the GTFS format of public transport information. Finally, we describe XMPP shortly.

Our approach has the most similarities with recent ideas on tracking public transport vehicles. The authors in [6] propose a bus arrival time prediction solution based on collecting sensor data by bus passengers. The proposed system uses movement statuses, audio recordings and mobile celltower signals to identify the bus and its actual position. The authors in [7] propose a method for public transport tracking based on accelerometer and GPS data collected on the users' mobile. EasyTracker [8] provides an inexpensive solution for real-time public transport tracking and mapping based on GPS sensor data collected by smart phones in public transport vehicles. It also offers arrival time prediction. The focus of these approaches is on data collection, since our focus is on provisioning services updated in real-time based on crowd collected data.

978-1-4799-0543-0/13/$31.00 ©2013 IEEE

GTFS [1] is used to represent public transport data for various operators around the globe. The GTFS database consists of comma delimited text files which describe the following GTFS feed elements. *Agency:* who provides the transit data; *Routes:* a route groups trips as a single service offered to passengers; *Stops:* individual locations where vehicles pick up or drop passengers; *Stop times:* vehicle arrival and departure times from the viewpoint of an individual stop; *Calendar:* weekly schedule of the service; and *Trip:* a sequence of two or more stops for each route that occurs at a specific time. In order for us to offer a competitive service even without participatory users we will also use static GTFS data as a basic service, which is provided from day zero of the application's launch.

XMPP [5] is an open technology for real-time communication using XML (Extensible Markup Language) [9] message format. XMPP allows sending of small information pieces from one entity to another in quasi real-time. It has several extensions, like multi-party messaging [10] or notification service [11]. This latter realizes a publish/subscribe (pubsub) communication model [12], where publications sent to a node are automatically multicast to the subscribers of that node. Furthermore, collection nodes [11] can be used to easily manage subscriptions through aggregate notifications. XMPP is well established and widely used in instant messaging services, like Google Talk [13] or Facebook Chat [14]. We have also chosen XMPP and the publish/subscribe communication model as the core building elements of our transit service architecture.

III. XMPP-based Live Transit Feed Design

In [4], we proposed a generic open architecture based on the XMPP protocol for mobile crowdsensing. We reuse and adapt that architecture in our real-time transit feed service design.

A. Requirements

Beyond the basic requirements we discussed in [4], such as extensible information model; decoupling between producers and consumers; unifying open architecture, the key success factor for crowdsensing applications is their ability to attract contributors. We think, that these applications must offer minimum similar services from their launch as those available already on the market to attract contribution. Hence, combining static GTFS and real-time crowd-sensed data offer immediate service and the quality of experience improves as the contribution level from the crowd increases. In this work, we investigate and discuss an application scenario where static GTFS data is combined with mobile crowdsensing.

B. Model

In order to limit the communication overhead for mobile users, we mapped the GTFS database into an XMPP node structure, which would enable fine grained selection of elementary feeds of possible interest by the users. Fig. 1 shows the pubsub nodes for content filtering in a transit feed. The transit information is structured into two views for the users, like route based and stop based. The route based view corresponds to the scenario, when the user would like to receive information related to specific trips (vehicles), while the stop based view corresponds to the scenario, when the user is interested in forthcoming arrivals at given stops (locations). The Trip nodes of the structure will receive live event updates, while other nodes would only contain references to relevant trip nodes.

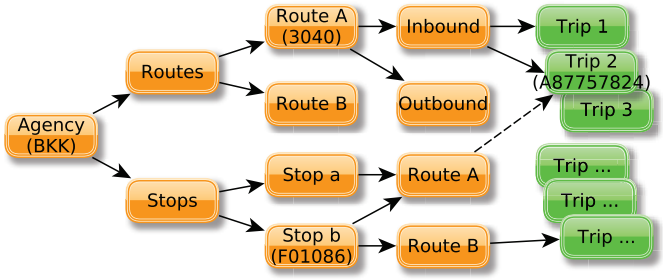


Fig. 1. Node Structure of our GTFS Feed Model

As an example (see the bracketed labels in Fig. 1 and Fig. 2), a node with ID "BKK-Routes-3040-in-A87757824" contains live updates to Trip=A87757824, inbound, Route=3040 of Agency=BKK. On the other hand, information related to Route 3040 (like full name, short name, head-signs, stops, etc.) is found as persistent content in "BKK-Routes-3040", while all currently active trips to inbound direction are found in "BKK-Routes-3040-in" node. Similarly, "BKK-Stops-F01086" node contains static information related to the stop in its headline (e.g., name, GPS coordinates, etc.) and the list of routes this stop belongs to. The route node, however, contains one entry per trip with the trip's ID as defined above.

C. Architecture

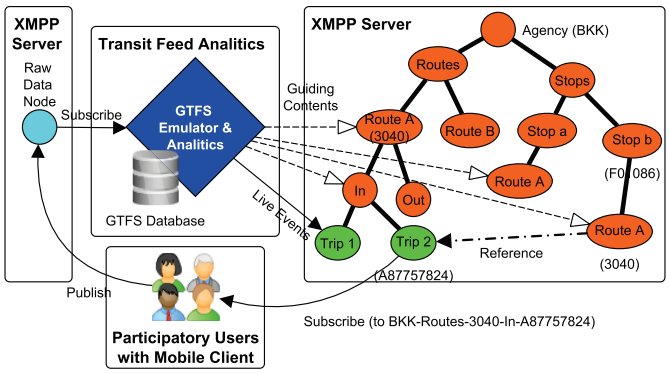


Fig. 2. Architecture for Live Transit Feed Service

Additional to a standard XMPP server, our architecture consists of a GTFS Emulator, analytic module(s) and a mobile client application (see Fig. 2). The latter one is based on a standard XMPP stack (e.g., ASmack [15]) which offers a user interface to navigate through the GTFS contents semi automatically and enables live feedback/data from the user. Event publication and feedback/data reception can be handled either by the same or by separate XMPP servers (the latter case is shown in Fig. 2). The GTFS Emulator must be in place to send transit information based on the static GTFS Database in the absence of user feeds. Thus, the GTFS Emulator provides

the day zero service in our scenario. On the other hand, the analytic module is responsible for the business logic offered by the service.

For our current performance analysis, we assume, that clients will navigate through the pubsub node structure shown in Fig. 1 and receive events accordingly. Moreover, we assume a pass-through analytic module, which passes through the user live feed directly to the corresponding content node.

IV. Evaluation

A. Tools

In all of our measurements we used some basic tools to load and measure the characteristics of an imaginary crowd transit service based on XMPP servers. In our setup, we used a GTFS emulator plus active and passive users.

GTFS Emulator: We developed a standalone GTFS emulator, which sends GTFS stop events into the XMPP server from a time stamped, ordered event list generated directly from a GTFS database. We mapped the static schedule of the agency to the crowd service with the emulator.

Passive Users: We set the load of our XMPP server by changing the number of passive user subscriptions to different pubsub nodes.

Active Users: Active users publish measurement messages to the XMPP nodes they are subscribed to, at every 100 msec. They measure the elapsed time between publishing and receiving their messages. We define the service round-trip-time (RTT) as the elapsed time measured by such active users. The RTT is a quality of service (QoS) parameter of the system, as it describes the time it takes for a live update to get to the subscribers.

B. XMPP Server

We used the Erlang Jabber/XMPP daemon (ejabberd) [16] as our XMPP server. We run the server on an AMD Athlon K9 Dual Core Processor 5050e hardware at 2600MHz with 2Gbyte RAM and 3.8.0-19 Linux kernel.

Our measurement setup consists of a single server architecture (see Fig. 3), where only one ejabberd server was used to carry the load. We measured the RTT for symmetric multiprocessing (SMP) disabled and enabled in the ejabberd server.

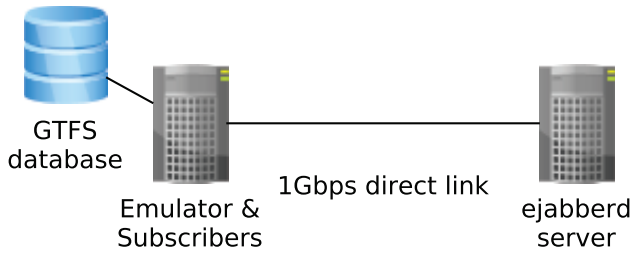


Fig. 3. Measurement Scenario

C. Methodology

We used active users to measure service RTT (see above). We were interested how a commodity XMPP server can handle the task of serving various number of users under the GTFS emulator's load.

We picked a busy hour (7am to 8am, weekday) from the GTFS database of Budapest's transit operator (BKK) as source to the emulator. In the event trace, 96.6% of the events belonged to bursts (GTFS databases show this characteristic as most of the arrivals are scheduled at solid minutes) with the rest spread evenly between these bursts. More specifically, the trace contained 60 bursts of approximately 1,000 stop arrival events per minute. Due to the back-to-back bursts of the stop events, we increased the playback speed of the emulator with a factor of three, without affecting the characteristics of the trace, resulting in a burst inter arrival time of 20 sec.

In order to avoid initial transients, we left out the first 5 bursts of each measurement and collected statistics for the remaining 55 bursts. We used Student's t-distribution to estimate a 95% confidence interval for the sampled mean of RTT. The active measurement clients sent probe messages at every 100 msec. The passive clients subscribed to all trip content channels (e.g., 10 passive users generated a load of $3 \times 10 \times 1,000 = 30,000$ events to send out per minute for the XMPP server).

Additionally, active users sent 600 measurement messages per minute, which were also sent to all subscribers. Therefore 3600 messages/minute load corresponds to a single active user in the system.

In our measurement setup, we made sure that none of the active or passive users, nor the emulator be the bottleneck, but only the ejabberd server after certain load. For this reason, we used multiple machines for generating active and passive loads and GTFS events.

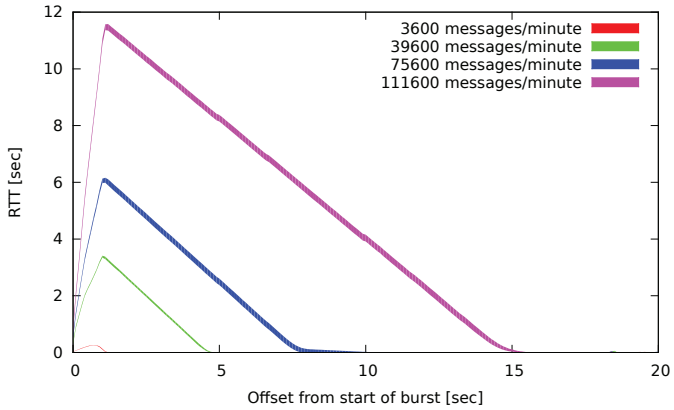
D. Results

We established basic estimates on how much load a not too new commodity hardware equipment can handle. Fig. 4 shows results for both SMP *on* and *off* states.

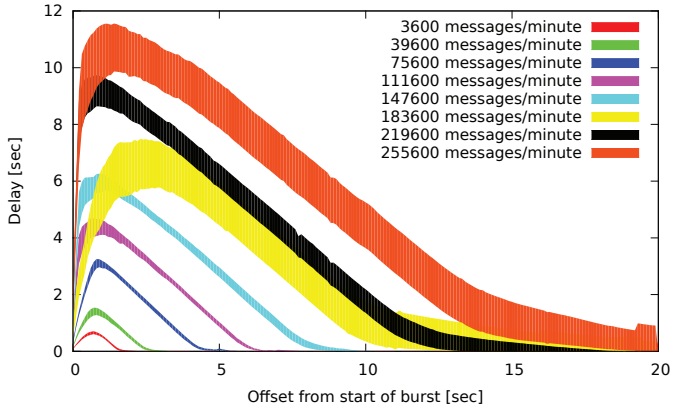
Along the x axes we show the time offset, when the active measurement message was sent to the XMPP server. The time offset is synchronized to the start of the corresponding burst of the GTFS emulator. Because of the three times playback speed of the emulator, bursts are started with a 20 sec inter-arrival time. If the time offset plus the experienced delay went over 20 sec, then the server could not cope with the offered load.

We used memory databases in the ejabberd server and did not store any pubsub messages persistently. The plots show 95% confidence intervals measured over 55 bursts. Fig. 4a shows, that our server, without multiprocessing, could carry over 100,000 messages per minute. With enabling multiprocessing on our dual core hardware, the server was able to cope with even a doubled load (see Fig. 4b). However, the higher load regions resulted in higher variance in the measured delay. It must also be noted that above 200,000 messages per seconds, the system becomes unstable, as the delays overlap across bursts.

This means – assuming that a single passive (subscriber but not publisher) user is interested in no more than 10 trips at any time, and that live updates are limited by the service logic to a maximum of one update per trip per minute – that one can launch an XMPP-based static plus live transit feed service with a commodity dual-core 2.6GHz PC with 2Gbyte RAM for 20,000 simultaneously on-line passive users.



(a) SMP OFF



(b) SMP ON

Fig. 4. RTT Measurements with 95% Confidence Interval

V. Summary

We investigated, how an incremental real-time transit feed service based on crowdsensing could be realized over commodity of the shell hardware PCs (COTS-PCs), standard protocols (XMPP), open source servers (ejabberd) and publicly available de-facto standard GTFS databases. We measured a single server setup to estimate baseline performance figures. We developed and used a GTFS emulator to estimate the load of the day zero service. In order to load the system, we attached subscribers to the transit feed service. We used active publishers (like participatory users) to measure the delay characteristic of the server. We have shown that a dual-core AMD Opteron 2.6 GHz PC with 2 Gbyte RAM can serve at best 200,000 transit feed messages per minute which can be mapped to about 20,000 on-line users.

Our future plan is to measure the performance of our system in case of using a cluster of XMPP servers.

Acknowledgment

The publication was supported by the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project. The project has been supported by the European Union, co-financed by the European Social Fund. This work has been partially supported by the KIC ICTLabs under the activity 13064 CityCrowdSource of the action line Digital Cities. Károly Farkas has been partially supported by the Hungarian Academy of Sciences through the Bolyai János Research Fellowship.

References

[1] Google Inc., "General Transit Feed Specification Reference." [Online]. Available: https://developers.google.com/transit/gtfs/reference/

[2] R. Ganti, F. Ye, and H. Lei, "Mobile Crowdsensing: Current State and Future Challenges," *IEEE Communications Magazine*, pp. 32–39, Nov. 2011.

[3] R. L. Szabo and K. Farkas, "Publish/Subscribe Communication for Crowd-sourcing Based Smart City Applications," in *Proceedings of the 2nd International Conference of Informatics and Management Sciences (ICTIC 2013)*, K. Matiasko, A. Lieskovsky, and M. Mokrys, Eds., Mar. 2013, pp. 314–319.

[4] ——, "A Publish-Subscribe Scheme Based Open Architecture for Crowd-sourcing," in *Lecture Notes in Computer Science 8115: Proceedings of 19th EUNICE Workshop on Advances in Communication Networking (EUNICE 2013)*. Springer, Aug. 2013, pp. 287–291.

[5] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Core," RFC 6120 (Proposed Standard), Internet Engineering Task Force, Mar. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6120.txt

[6] P. Zhou, Y. Zheng, and M. Li, "How Long to Wait?: Predicting Bus Arrival Time with Mobile Phone based Participatory Sensing," in *Proceedings of the Tenth International Conference on Mobile Systems, Applications, and Services (MobiSys 2012)*, Jun. 2012.

[7] A. Thiagarajan, J. Biagioni, T. Gerlich, and J. Eriksson, "Cooperative Transit Tracking Using Smart-phones," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys 2010)*, Nov. 2010, pp. 85–98.

[8] J. Biagioni, T. Gerlich, T. Merrifield, and J. Eriksson, "EasyTracker: Automatic Transit Tracking, Mapping, and Arrival Time Prediction Using Smartphones," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys 2011)*, Nov. 2011, pp. 1–14.

[9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (XML) 1.0 (fifth edition)," W3C, W3C Recommendation REC-xml-20081126, Nov. 2008. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/

[10] P. Saint-Andre, "XEP-0045: multi-user chat," XMPP Standards Foundation, Standards Track XEP-0045, Feb. 2012. [Online]. Available: http://xmpp.org/extensions/xep-0045.html

[11] P. Millard, P. Saint-Andre, and R. Meijer, "XEP-0060: Publish-subscribe," XMPP Standards Foundation, Draft Standard XEP-0060, Jul. 2010. [Online]. Available: http://xmpp.org/extensions/xep-0060.html

[12] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[13] Google Inc., "Google Talk for Developers: Open Communications," Mar. 2013, last updated May 15, 2013. [Online]. Available: https://developers.google.com/talk/open_communications/

[14] Facebook Inc., "Facebook Chat API," 2013. [Online]. Available: http://developers.facebook.com/docs/chat/

[15] aSmack Contributors, "aSmack API." [Online]. Available: https://github.com/Flowdalic/asmack/

[16] ejabberd Community, "ejabberd – Distributed Fault-tolerant Jabber/XMPP Server in Erlang," Aug. 2013. [Online]. Available: http://www.ejabberd.im/