Performance Evaluation of a Live, Crowdsensing Based Transit Feed Service Architecture

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

***Abstract*—Taking public transportation is an efficient and environmentally sound way of traveling in most of the cities. Unfortunately, the static schedule information of the public transport vehicles, available at the stops, on the web or in some special format like GTFS (General Transit Feed Specification), usually does not reflect the actual traffic situation. However, real-time traffic updates require the gathering of an immense amount of tracking data. Mobile crowdsensing, via the mobile devices of the crowd, can offer a cheap and efficient way for collecting such data. Nonetheless, for motivating active participation some day zero service must be provided to the users. In this paper, we discuss how to realize a live public transport information service extending a static timetable based on GTFS data, as the day zero service, using the power of the crowd for data collection. We detail the design of such a service implemented by an XMPP (Extensible Messaging and Presence Protocol) based mobile crowdsensing architecture and evaluate its performance. We show how we can build a scalable service architecture even with commodity hardware to handle thousands of users.**

***Keywords*—Crowdsensing, Public transport, GTFS, Publish/subscribe, XMPP**

I. Introduction

Commuting has become part of our daily routine which results in spending a significant part of our time with traveling. Public transportation offers a viable alternative for commuters, especially in crowded cities, reducing private vehicle usage, fuel consumption, environmental pollution and alleviating traffic congestion. However, it is important for the passengers to have accurate information about the arrival time at the stops of the public transit vehicles. Otherwise, unexpectedly long waiting times can frustrate passengers and divert them from using public transport. It is even better if some other extra information about the arriving vehicle, for instance the congestion level or baby buggy friendliness, is also provided, making travel planning easier.

Fortunately, most transit operators of big cities make their timetables or transit feeds freely available for the passengers or third party application developers. The General Transit Feed Specification (GTFS) [1] – originally developed by Google – evolved into the de-facto standard to exchange transit feeds publicly. However, the widely used GTFS format enables only the exchange of static transit information (e.g., departure schedules, travel time, operating hours), which does not reflect live traffic conditions. Remedy to live updates is the relatively new real-time extension to GTFS [2], which relies on operators to invest into a real-time fleet tracking and communication infrastructure to be able to provide live service updates. Unfortunately, nowadays very few operators offer such services due to the necessary investments.

Another approach to live updates is using mobile crowdsensing[1] [3] and let the crowd collect the information required to extend the basic timetable service. In this case, voluntary passengers sense and send live service updates (e.g., delays, congestion, hazard, etc.) to a service provider via their smart-phones. The service provider then aggregates, cleans, analyzes the collected data, and derives and disseminates the live updates to the users. For sensing, the built-in and ubiquitous sensors of the mobile phones can be used either in a participatory or an opportunistic way depending on whether data collection happens with or without user involvement. The contribution of every traveler can be useful. Hence, passengers waiting for a trip can report the line number with a timestamp of every arriving public transport vehicle at a stop during the waiting period. On the other hand, on-board passengers can send actual location information of the moving vehicle periodically and report events of arrival at/departure from the stops.

Although the participatory sensing based approach is a viable alternative, it faces several challenges. The basic challenge, similarly to other crowdsensing based services, is how passengers can be motivated to participate in data collection. We believe, that *a day zero service*, which is provided from the beginning and improved, based on the crowd-sensed data, following incremental service development, can be an appropriate incentive. Along this line, as a sequel work to [4] and [5], *we propose and investigate a scenario, where a static transit feed as the day zero service is improved and updated with live service updates from participatory users*. We show how such a scenario can be implemented by the Extensible Messaging and Presence Protocol (XMPP) [6]. We evaluate different XMPP server setups under load. We present measurement results showing that even commodity PCs are able to carry on with the load and handle several thousand users, and that XMPP server clustering can remedy service scaling. We show, that with our architecture and a GTFS emulator static and live service updates can easily be combined introducing incremental service improvements based on mobile participatory users.

The rest of the paper is structured as follows. In Sec. II we

[1]We use the terms *crowdsensing*, *crowdsourcing* and *participatory sensing* interchangeably in this paper.

978-1-4799-4615-0/14/$31.00 ©2014 IEEE

review related work. We introduce our design in Sec. III. We present our measurement based evaluation results for different service setups in Sec. IV. Finally, we draw conclusions in Sec. V.

II. Related Work

In this section, first we review mobile participatory sensing based transit tracking applications. Next, we discuss GTFS as the de-facto format of transit feed exchange. We explain, why we build our transit feed service also on this specification. Finally, we shortly describe XMPP, as our choice of architecture.

A. Transit Tracking Using Mobile Phones

Our approach has the most similarities with recent ideas on transit tracking systems. The authors in [7] propose a bus arrival time prediction system based on bus passengers' participatory sensing. The proposed system uses movement statuses, audio recordings and mobile celltower signals to identify the vehicle and its actual position. The authors in [8] propose a method for transit tracking using the collected data of the accelerometer and the GPS sensor on the users' smart-phone. EasyTracker [9] provides a low cost solution for automatic real-time transit tracking and mapping based on GPS sensor data gathered from mobile phones which are placed in transit vehicles. It offers arrival time prediction as well.

These approaches focus on the data (what to collect, how to collect, what to do with the data) to offer enriched services to the users. However, our focus is on how to introduce such enriched services *incrementally*, i.e., how we can create an architecture and service model, which allows incremental introduction of live updates from participatory users over static services that are available in competing approaches. In our view, our contribution complements the ones with focus on analytics and automated transit context detections.

B. GTFS and GTFS-realtime

GTFS [1] is used to represent public transport data for various operators around the globe. Google Maps, for example, uses GTFS data sources to plan a trip taking public transport in major cities. In our transit feed service, we rely on the GTFS data model and terminology, hence we briefly describe here the most important and mandatory data structures of a GTFS feed. 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 (see below) as a single service offered to riders (a.k.a. lines); *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.

GTFS-realtime (GTFS-rt) [2] is an extension to the GTFS specification. It defines the feed format which allows public transportation agencies to provide live updates about their fleet to application developers. The specification defines the following feed types: *Trip updates:* delays, cancellations or changes to trips; *Service alerts:* unforeseen events affecting a station, route or entire network; *Vehicle positions:* location and congestion states related to vehicles. In order for an agency to track its fleet and provide realtime GTFS feeds, it has to invest into a costly infrastructure, both hardware and software. Today, only a few Google partner agencies provide GTFS-rt services.

On the other hand, there exists a wide variety of third party applications for smart-phones, e.g., OneBusAway [10], which offer access to GTFS transit feeds in various presentations. Common to most of them is that their service is static without live updates. 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.

C. XMPP

XMPP [6] is an open technology for real-time communication using XML (Extensible Markup Language) [11] 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 [12] or notification service [13]. This latter realizes a publish-subscribe (pubsub) communication model [14], where publications sent to a node are automatically multicast to the subscribers of that node. Moreover, collection nodes [13] can be used to easily manage subscriptions through aggregate notifications. XMPP is well established and widely used in instant messaging services, like Google Talk [15] or Facebook Chat [16].

On the other hand, beyond implementing instant messaging, XMPP can also be used for other service architectures. For instance, the authors in [17] proposed to bring XMPP into the Internet of Things embedded systems. They concluded with experiments that XMPP can be minimized to run on resource constrained devices and being able to communicate with full fledged clients. Based on their results it will be possible to not only bring smart-phone based sensing into the common platform of XMPP, but also smart objects. Another example is the Social Backbone (SBone) [18], [19] which provides an open architecture based on the Jabberd2 [20] XMPP server. It allows personal devices to share their resources and state with each other, seamlessly and securely, using a social network for authentication, naming, discovery and access control. These approaches show similarities with our use-case and demonstrate the applicability of XMPP beyond the basic instant messaging scenario. So, we have also chosen XMPP as the core building element of our transit service architecture.

III. Design of the XMPP Based Live Transit Feed Service

We discussed the use of an XMPP based generic open architecture for mobile participatory sensing in [5]. In our service design, we consider that architecture and adapt it to our live transit feed service.

A. Requirements for Developing Crowdsensing Based Services

We discussed the basic, high level requirements for crowdsourcing based service development in [5]. These are: i) unifying open architecture; ii) extensible information model; and iii) decoupling between producers and consumers. However, the key issue for participatory sensing applications is how they can attract user contributions.

To capture interest and motivate contribution, we believe that such applications must offer a day zero service. This means that the new application must provide at least similar service level and user experience to the already available services from the launch day of the application, otherwise nobody will use it. For instance, a static public transportation timetable can be offered as the immediate day zero service and the quality of experience of this service may be improved with live crowd-sensed information as more and more users start to participate. In the following, we discuss and examine such a use-case scenario where the day zero service, based on static GTFS data, is combined with mobile crowdsensing.

B. The Publish-Subscribe Model for GTFS Feeds

We turned the GTFS database into an XMPP node hierarchy to avoid unnecessary communication overhead. This node structure facilitates searching and selecting transit feeds according to user interest. The pubsub node model for content filtering in a transit feed (see Sec. II-B for the terminology) is depicted in Fig. 1.

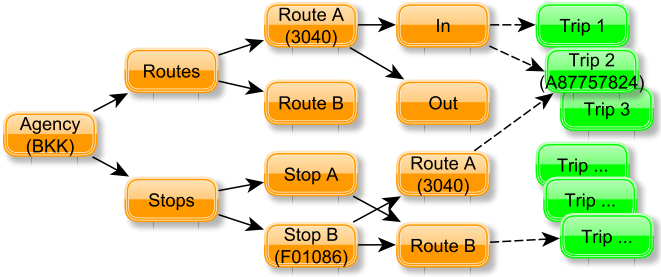


Fig. 1. Publish-Subscribe Model for GTFS Feeds

Transit information and real-time event updates are handled in the *Trip* nodes at the leaf level. The inner nodes in the node hierarchy contain only persistent data and references relevant to the trips. The users can access the transit data via two ways, based on *routes* or *stops*. When the user wants to see a given trip (vehicle) related traffic information the route based filtering is applied. On the other hand, when the forthcoming arrivals at a given stop (location) are of interest the stop based filtering is the appropriate access way.

For instance, the leaf node with trip ID "BKK-Routes-3040-In-A87757824" (cf. the bracketed labels in the nodes of Fig. 1 and Fig. 2) handles the transit feed and its real-time updates related to Trip 2 in the inbound direction belonging to Route A of Agency BKK (operator at Budapest, Hungary). On the other hand, node "BKK-Routes-3040" stores persistent transit information with regard to Route A (e.g., route name, short name, stops, head-signs), since references to all the currently active inbound trips are found in node "BKK-Routes-3040-In". Similarly, node "BKK-Stops-F01086" stores persistent data with regard to the given stop (e.g., stop name, GPS coordinates) and lists the routes this stop is part of. Furthermore, the trip ID of every active trip is listed in the route node.

C. Architecture of the Live Transit Feed Service

Our service architecture (see Fig. 2) consists of a standard XMPP server or federation of XMPP servers, a GTFS Emulator, analytics module and a mobile client application. The mobile application is using a standard XMPP stack (e.g., ASmack [21]). This provides a semi automatic navigation of the GTFS data through a user interface and makes possible to send report/feedback or real-time sensor data (e.g., crowdedness information, vehicle arrivals at the stops) to the server. Either the same or separate XMPP servers (the latter is shown in Fig. 2) can be used to handle the published events and the collected data/reports. When user feeds are not available the GTFS Emulator, using the static GTFS Database, publishes transit information. Hence, the day zero service is provided by the GTFS Emulator in our case. The business logic of the service, e.g. stop event detection or estimating vehicle arrivals at the stops, is handled by the analytics module.

In our performance analysis, we use the pubsub node structure depicted in Fig. 1. The clients will receive traffic updates according to their subscribed channels. Moreover, we implement only a simple analytics module that disseminates live feed, created by the GTFS Emulator in our experiments, into the appropriate pubsub channels.

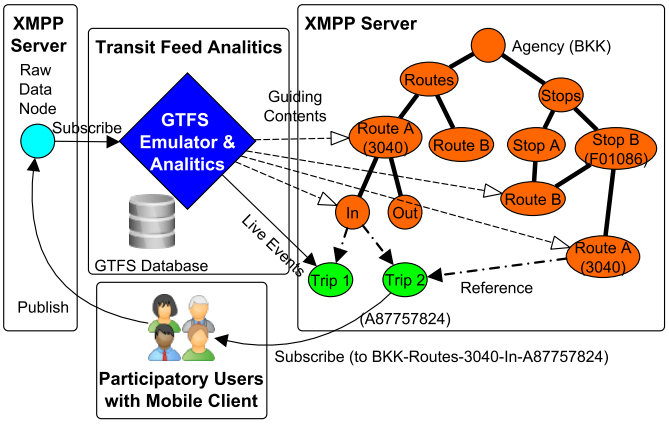


Fig. 2. XMPP Based Pubsub Architecture for Live Transit Feed Service

IV. Evaluation

A. Measurement Components

In our measurements, we used some basic components to load and measure the characteristics of an imaginary crowd-sourced transit feed service. In our setups, we used XMPP Server(s), GTFS Emulator, Active and Passive Clients.

- *XMPP Server(s):* We used the Erlang Jabber/XMPP daemon (ejabberd) [22] as our XMPP server(s). We ran the server(s) on AMD Athlon K9 Dual-Core Processor 5050e hardware at 2600 MHz with 2 GByte RAM and 3.8.0-19 Linux kernel.
- *GTFS Emulator:* We developed a standalone GTFS Emulator, which sends GTFS stop events into the XMPP server(s) from a time stamped, ordered event list generated directly from a GTFS Database (see

Sec. II-B). We mapped the static schedule of the agency to the crowd service with this Emulator.

- *Passive Clients:* We set the load of our XMPP server(s) by changing the number of Passive Client subscriptions, emulated by virtual machines, to different pubsub nodes.
- *Active Clients:* Active Clients, emulated by virtual machines, publish measurement messages to the XMPP server(s) they are subscribed to at every 100 msec. They measure the *Service Time* which is the elapsed time between publishing and receiving a measurement message.

B. Measurement Setups

We investigated three different measurement setups (see Fig. 3, black solid lines - physical architecture, colored dashed lines and colored device names - logical architecture):

1) Our first setup consisted of a single machine (beyond the GTFS Emulator, GTFS Database and an Ethernet Switch), where only one XMPP server (see the blue dashed lines and the Server in Fig. 3) was used to carry the load. We used this setup for the baseline measurements.
2) In our second setup, we used a cluster of XMPP servers to investigate scalability and extended the single server setup with two additional servers. We organized the three servers into a hierarchy (see the green dashed lines and the Master, Slave 1, Slave 2 in Fig. 3) where the Master server received and replicated the transit feed input from the Emulator while the two Slave servers carried the load of subscribers.
3) In the third setup, we used a cluster of three XMPP servers again (see the red dashed lines and the Server 1, Server 2, Server 3 in Fig. 3), but this time with application level load sharing. Thus, we do not have any dedicated server rather the Emulator distributes the load among the servers fulfilling equal role.

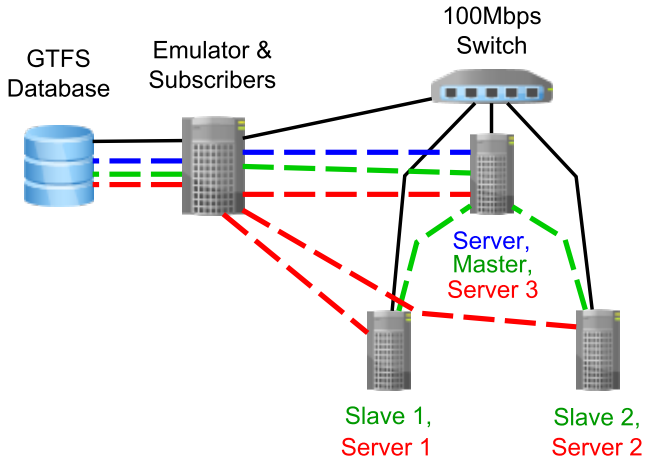


Fig. 3. Measurement Scenarios

C. Measurement Methodology

We used active clients to measure the Service Time (see above). We were interested in how XMPP server(s) running on commodity PC(s) 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 (capital of Hungary) transit operator (BKK) as the 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. We increased the playback speed of the Emulator with a factor of three, without affecting the characteristics of the trace, resulting in a burst interarrival time of 20 sec.

We used Student's t-distribution to estimate a 95% confidence interval for the sampled mean of the Service Time. The active measurement clients sent probe messages at every 100 msec. The passive clients subscribed to all the trip content channels (e.g., 10 passive users generated a load of $3 \times 10 \times 1,000 = 30,000$ events to be sent out per minute by the XMPP server(s)). Additionally, active clients sent 600 measurement messages per minute, which were also sent to all subscribers. Therefore, 3600 message/min load corresponds to a single active client in the system.

In our measurement setups, we made sure that none of the active or passive clients, nor the Emulator be the bottleneck, but only the XMPP server(s) after certain load. For this reason, we used multiple machines to generate the active and passive load and GTFS events.

D. Results: Single Server Setup

With the single server setup, we established baseline measurements. Along the x axis the time offset is depicted, when the active measurement messages were sent to the XMPP server. The time offset was synchronized to the start of the corresponding burst generated by the GTFS Emulator. Since the three times playback speed of the Emulator, bursts were started with a 20 sec interarrival time. We used memory databases in the XMPP server.

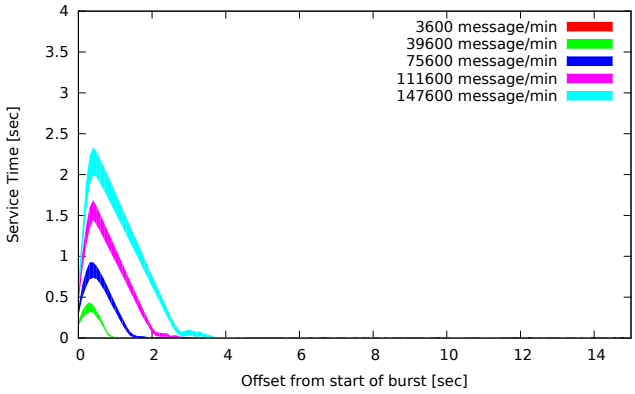


Fig. 4. Service Time with 95% Confidence Interval in the Single Server Setup

Fig. 4 shows the Service Time with 95% confidence interval measured over 55 bursts under different loads. We can

see that a single server can handle about 150,000 messages per minute with a Service Time not greater than 2.3 seconds even at the peak of the bursts (in case of higher load we already experienced data losses).

E. Results: Server Cluster Setup with One Master and Two Slaves

With this setup, we were interested to investigate how our service can scale in hardware without application support. The clustering mechanism in ejabberd enables the architecture itself to distribute the load transparently. Fig. 5 shows the Service Time with 95% confidence interval measured at the Slave servers (the Master did not handle subscriptions in this setup). Note, that the plots depict the total load of the system and not the per server load. With regard to this setup we can observe the following points:

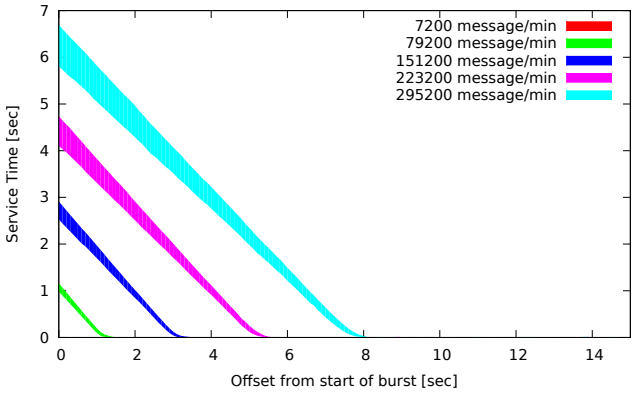
- The load was almost evenly distributed between the Slaves.
- Each of the slaves was able to serve as many messages as the single Server in the previous setup.
- The Master server is not a bottleneck if no subscribers are attached to it.
- The two stages of server processing almost tripled the Service Time under the same per server load (see Fig. 5a and 5b).

Basically, with the additional cost of a Master server, the load handled by this architecture can linearly be increased by adding Slaves to the system with the expense of a higher Service Time. It remains to be further investigated (our future plans), how many more Slaves can be added to a Master.

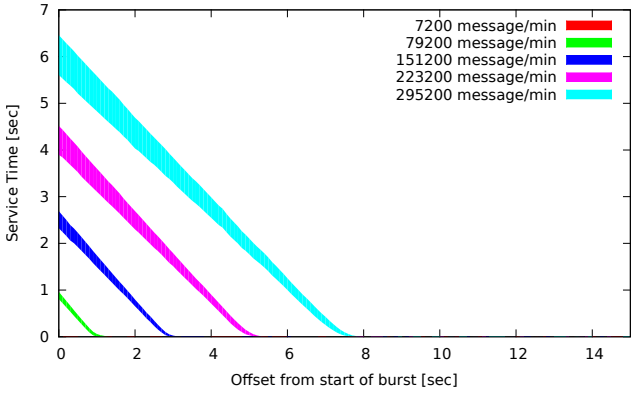
F. Results: Server Cluster Setup with Application Layer Load Sharing

In order to get rid of the extra delay introduced by the Master server in the previous setup, we can create an application layer load sharing scheme, where XMPP content nodes are uniformly mapped to different XMPP servers by some deterministic method. In this setup, our Emulator randomly mapped each new trip content node to one of the three available XMPP servers. The measurement results of this scenario are shown in Fig. 6. Note again, that the plots depict the total load of the system and not the per server load. With regard to this setup we can observe the following points:

- The Service Time characteristics are similar to the single server setup (except on Server 3).
- There are differences between the servers, which might reflect the uneven load distribution from the Emulator. This can be explained by the fact, that we mapped trips to different servers, however, there are significant differences in event rates within trips.
- These results do not reflect the overhead incurred at the Emulator to share the load. It also remains to be further investigated.



(a) Master → Slave 1



(b) Master → Slave 2

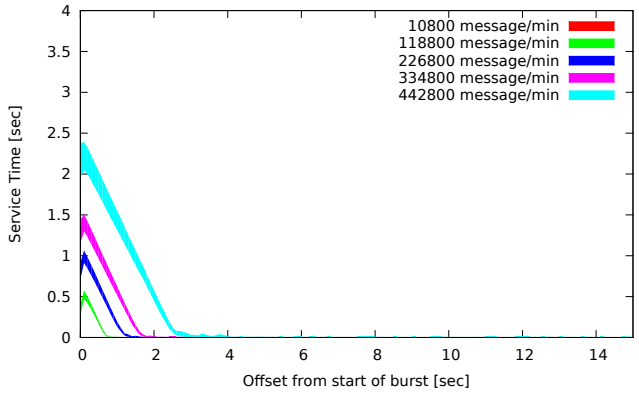
Fig. 5. Service Time with 95% Confidence Interval in the Server Cluster Setup: One Master and Two Slaves

V. Conclusions

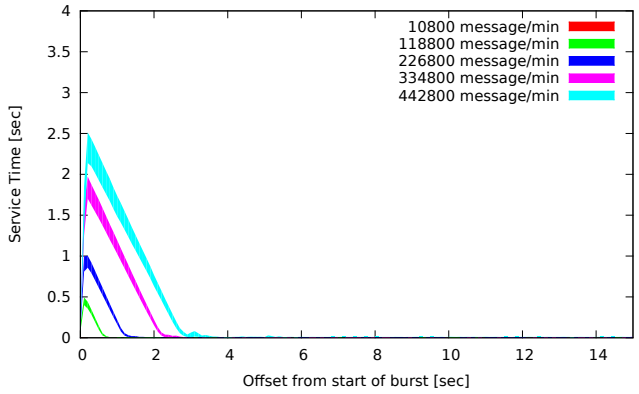
In this paper, we discussed how an incremental real-time transit feed service based on crowdsensing could be realized over commodity hardwares (PCs), standard protocols (XMPP), open source servers (ejabberd) and publicly available de-facto standard GTFS databases. We investigated the performance of our service architecture in case of 3 different setups, such as a single server, a server cluster with a Master and two Slaves, and a cluster of three servers with application layer support for load sharing. We have not discussed, yet how our results can be mapped to real user numbers of a live transit feed service. If we assume, that a single passive (subscriber) user is interested in no more than 10 trips at any time, and the live updates are limited to one update per trip per minute, then we can serve about 15,000 simultaneous on-line passive users with our commodity dual-core AMD Athlon 2.6 GHz PC with 2 GByte RAM keeping the Service Time below 2.5 seconds, or multiple of them selecting an appropriate server cluster architecture. As our future work, we plan to investigate further the server cluster setups, and use our prototype Android client application under development in the measurements to generate live feeds as real load.

Acknowledgment

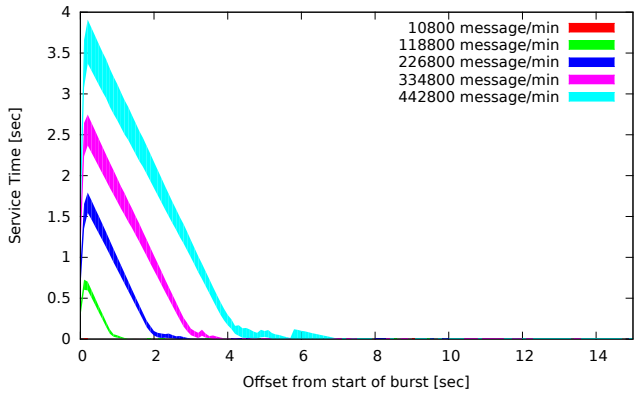
The publication was supported by the EITKIC_12-1-2012-0001 project, which is supported by the Hungarian Govern-



(a) Server 1



(b) Server 2



(c) Server 3

Fig. 6. Service Time with 95% Confidence Interval in the Server Cluster Setup: Three Servers with Application Layer Load Sharing

ment, managed by the National Development Agency, financed by the Research and Technology Innovation Fund and was performed in cooperation with the EIT ICT Labs Budapest Associate Partner Group (www.ictlabs.elte.hu). 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] ——, "General Transit Feed Specification Realtime." [Online]. Available: https://developers.google.com/transit/gtfs-realtime/

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

[4] 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.

[5] ——, "A Publish-Subscribe Scheme Based Open Architecture for Crowd-sourcing," in *Proceedings of 19th EUNICE Workshop on Advances in Communication Networking (EUNICE 2013)*. Springer, Aug. 2013, pp. 1–5, to appear.

[6] 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

[7] 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.

[8] 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.

[9] 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.

[10] OneBusAway Developers, "OneBusAway." [Online]. Available: https://github.com/OneBusAway/

[11] 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/

[12] 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

[13] 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

[14] 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.

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

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

[17] M. Kirsche and R. Klauck, "Unify to Bridge Gaps: Bringing XMPP into the Internet of Things," in *Proceedings of the 10th IEEE International Conference on Pervasive Computing and Communications Workshops, Work in Progress (PERCOM Workshops)*, Mar. 2012, pp. 455–458.

[18] P. Shankar, B. Nath, L. Iftode, V. Ananthanarayanan, and L. Han, "Sbone: Personal Device Sharing Using Social Networks," Technical Report, Rutgers University, Tech. Rep., 2010. [Online]. Available: http://www.cs.rutgers.edu/~iftode/SBone10.pdf

[19] P. Shankar, L. Han, V. Ananthanarayanan, M. Muscari, B. Nath, and L. Iftode, "A Case For Automatic Sharing over Social Networks," in *Proceedings of the First ACM SIGKDD International Workshop on Hot Topics on Interdisciplinary Social Networks Research (HotSocial 2012)*, Aug. 2012.

[20] Jabberd Community, "Jabberd," http://jabberd2.org/.

[21] aSmack Contributors, "aSmack API," https://github.com/Flowdalic/asmack/.

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