Cloud Based System for Detecting Children in Automobiles

Adnan Shaout
The Electrical and Computer Engineering Department,
The University of Michigan
Dearborn, Michigan USA
shaout@umich.edu

Brennan Crispin
The Electrical and Computer Engineering Department,
The University of Michigan
Dearborn, Michigan USA
bhcrisp@umich.edu

***Abstract*—Children and infants being left in cars on hot summer days is a major public health issue across the world. With modern sensor and connectivity technology, it should be possible to detect these scenarios by uploading raw data to the cloud. In the cloud, systems will be able to respond to this data and make judgements to notify proper response personnel. In this paper, we explore cloud technology to process updates from a sensor system and how it could be used to send notifications to prevent these tragic occurrences.**

***Keywords*—Connected Vehicles, Cloud Computing, MQTT, REST, SMS**

I. Introduction

With the multiplication of cheap sensors that can be added to automobiles and connected to the internet, it has become possible to address a major public health issue – ensuring that infants and small children left in automobiles do not succumb to heat during the summer months [6]. While sensors can detect the presence of children in the automobile, it is not always easy to process that data or to alert outside persons so that action can be taken. In this paper, a cloud based system will be designed that can incorporate with existing sensor systems using MQTT [1], process data, and send a notification to an emergency contact [2] who can respond to the event in a timely manner.

Due to the contractual nature of the MQTT and SMS interfaces, this project was built using the Waterfall methodology. As such, a requirements phase, design phase, implementation phase, and testing phase were passed through in that order and will be discussed through this paper. Additionally, a PSP process was followed to capture defect and timing information to be compared for future large software projects.

This paper will be organized as follows: Section 2. Prior Work is discussed, and previous solutions for detection and vehicle cloud connectivity are investigated. In Section 3, the requirements for the project are discussed, and in Section 4, the design is elaborated on. Specific implementation details are discussed in Section 5. In Section 6 the results of system testing and benchmarking are discussed. Finally, in Section 7 we discuss conclusions and further work.

II. Prior Work

In order to understand the state of detection for this issue, research was conducted both in academic and corporate domains for existing solutions for this problem. In Zarife et al [3], the authors explored a system to detect sound or movement in an automobile and use that to identify children in the car. In [4], Viksnins et al. demonstrate a solution using pressure and temperature sensors to detect an infant that has been left, while Rossi [5] displays a child seat designed to notify parents.

Additionally, a great deal of work has been done in how connected vehicles will connect to the cloud. Prarna et al [7] explore how MQTT will be the most protocol for connecting vehicles to the cloud. Additional work [8][9][10] has also shown MQTT to be a preferred protocol for lower power devices to connect and publish data to the internet.

Haberle et al [11] explore how connected cars can be connected to the cloud using existing technologies, and how to create a platform for prototyping telematics. Further, Bitam and Mellouk [12] examine how cloud computing will be part of the intelligent transportation system. Clearly, there is a great deal of possibility in connecting cars to the cloud, and there exists technology to do so and to detect the sorts of events this project seeks to address. However, there does not appear to have been research performed that will use the cloud as a platform for collecting this specific data and routing it to responders.

III. Requirements

The objective for this paper is to create a cloud-based system that can accept vehicle sensor data, detect potential emergency events – such as a child left in the vehicle, and notify emergency personnel. Figure 1 shows the overall system requirements. The system will need to provide an endpoint for sensor systems to push data to, a set of APIs that can be used to register sensors, users, and contact information, and a system for notifying emergency contacts when emergency events occur. The following will present a brief discribtion of proposed system needs:

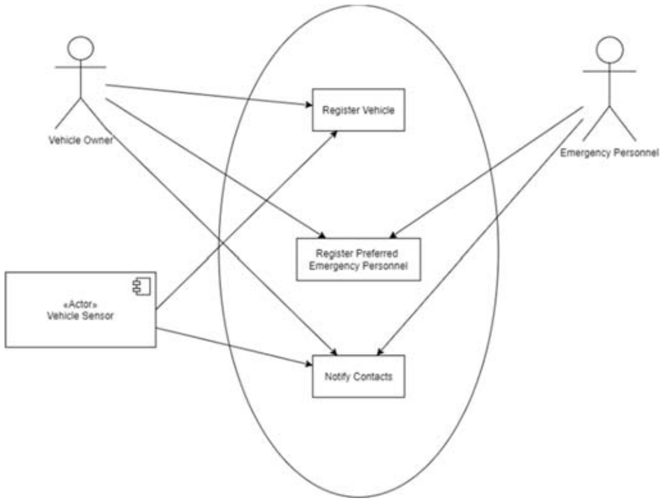

Figure 1. Overall System Requirements

A. *Registration APIs*

Since the system will need to be able to track and monitor vehicles and users interacting with it, it will be necessary to implement a set of Restful APIs [13] to add and remove contact and vehicle data as needed.

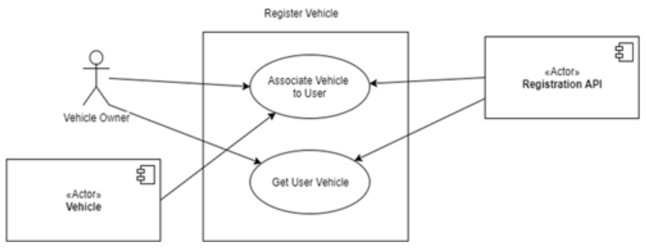

Figure 2. Registration API

The primary registration would require that the vehicle owner 'pair' with his vehicle, and then send in a request including the correct ID for that vehicle. The vehicle's system would send in a subsequent request with the user ID. Additionally, an API for registering contact info, which will be tied to a user ID will be required. Figure 2 shows the proposed registration API user case diagram.

B. *Detecting Emergency Events*

In order to detect emergency events, the system will need to provide a way for external sensors to upload data that can be processed. Since the sensors being used will employ MQTT technologies to communicate with cloud systems, it will be expected that this system will be able to connect to an MQTT broker and subscribe to data being pushed by each vehicle sensor. Figure 3 shows the detect emergency even user case diagram.

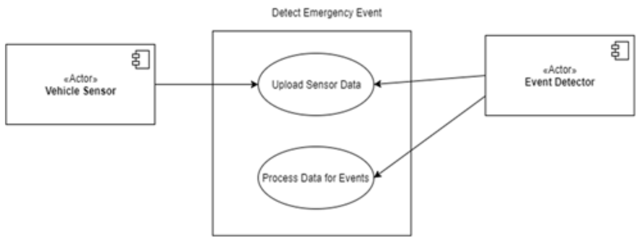

Figure 3. Detect Emergency Event

Before the system will be able to listen for events, it will be necessary for the user and vehicle associated to have registered with the APIs discussed above. Once a message is received, the system will parse it and extract event data to determine if a notification needs to be sent.

C. *Report Emergency Event*

When an emergency event is detected, it will be necessary for the system to send notifications to emergency personnel who can respond. Although there are many systems that emergency contacts may respond to, SMS is the most general and will be used for this system. Figure 4 shows the notification emergency contacts user case diagram.

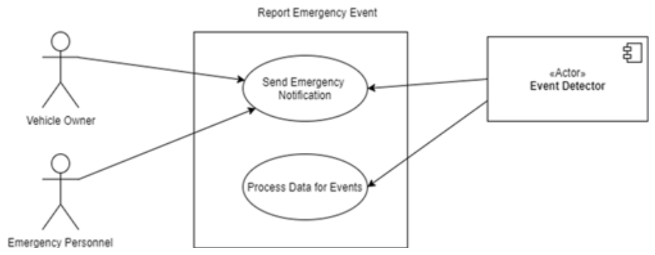

Figure 4. Notifying Emergency Contacts

D. *Non-Functional Requirements*

Since failure of this system could lead to death or injury, there are several non-functional requirements that are expected to be met. First, the time from an event being published to the MQTT broker to a notification being received by emergency personnel should not exceed 60 seconds. Second, service downtime should not exceed 0.001% of operation time.

Additionally, TLS will be the standard communication protocol for all web-based communication, and any API calls will be properly authenticated to ensure safety of private user data.

IV. Design

In this section, a brief overview of the design of the system will be provided, including major components, their relationships, and the data flow for each component. This system will contain five main components: a user/vehicle registration API, the event data stream, the event detector and processor, the contact registration API, and the send notification. Figure 5 shows the overall component relationships. The following will present a brief description for each of the five main components:

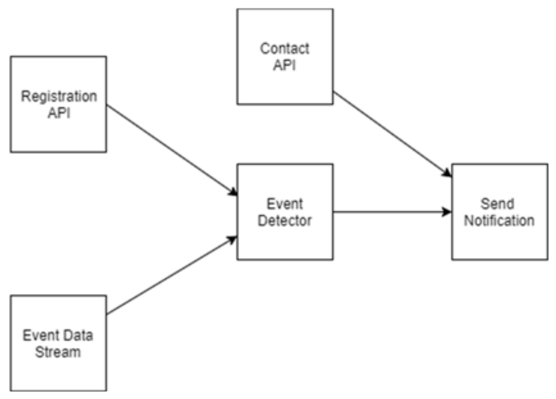

Figure 5. Overall Component Relationships

A. *User/Vehicle Registration*

This API will be supplied that will allow users and the vehicles to register themselves and to create associations that will be used for identifying contact information. The registration API will consist of a set of HTTP endpoints for creation.

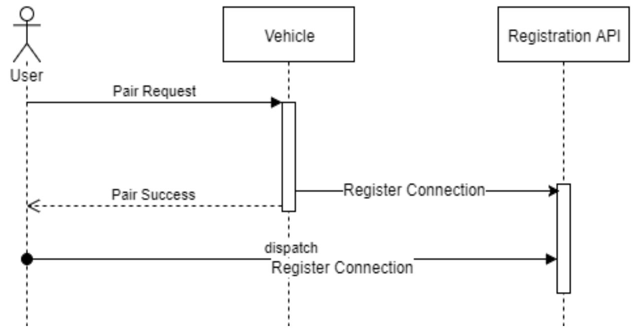

Figure 6. Flow Diagram for User/Vehicle Registration

The endpoints will receive the user ID and vehicle IDs, and when both have been received, will persist the new user and vehicle data as well as the association in a database. Figure 6 shows the flow diagram for User/Vehicle Registration.

B. *Event Data Stream*

In order to listen to events that are being created by vehicle sensors, the system will connect to an MQTT data stream. As a vehicle ID is added to the registration, the MQTT service will connect to the broker and subscribe to the vehicle ID created. As events are received, they will be parsed into data objects, evaluated, and then acted upon as shown in Figure 7.

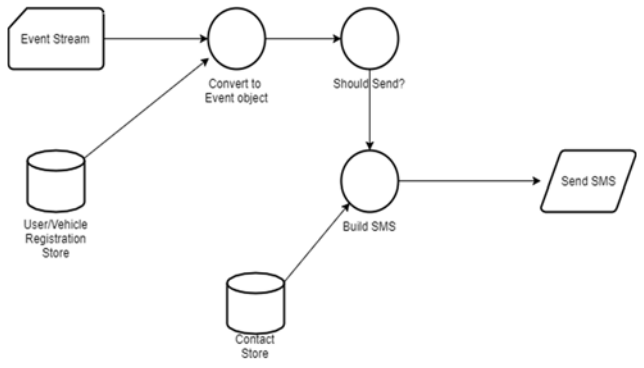

Figure 7. Data Flow Diagram for Event Detection

C. *Event Detector*

The Event Detector takes event data objects created by the event data stream processor, and evaluates them to emergency conditions. For this system, this will take a simple form of evaluating fields for event type and event data against a predetermined acceptable range. Any values outside that range will trigger a notification flow.

D. *Contact API*

Similar to the User/Vehicle Registration API, this will allow emergency contacts to be added for each user. An HTTP request will be received that contains a user ID and a contact's information. That data will then be persisted to the database.

E. *Send Notification*

Once an emergency event has been detected, the notification flow will be activated. In this flow, the contacts associated with the registered user will be retrieved from the database, and then an SMS message with the event information will be sent to each emergency responder in the list as shown in Figure 8.

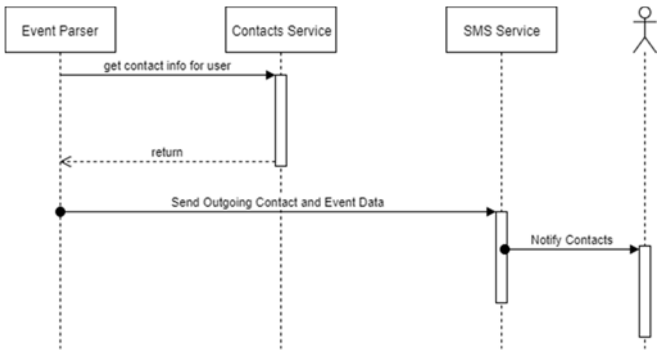
Event Parser
Contacts Service
SMS Service
get contact info for user
return
Send Outgoing Contact and Event Data
Notify Contacts

Figure 8. Flow Diagram for Sending Notifications

V. Implementation and Testing

This section discusses the software setup and configuration for the creation of the system.

A. *Server*

For this system, Spring and Spring Boot [15] were used to build the HTTP endpoints and to manage component interaction. This system, built on Apache, provides numerous services for developing server applications. The server is being run on locally on a 64 bit Windows machine with TCP and TSL ports open for connectivity.

B. *MQTT*

For the MQTT broker, Mosquito MQTT [16], provided by the Eclipse Foundation, was used. For testing purposes, a common web broker was used with a unique ID generated to keep publishers and subscribers separate from other users. SpringMQTT was used to manage connections locally to the broker, which allowed for faster development time and easier integration with the server infrastructure.

C. *SMS Messages*

For outgoing SMS messages, a library called Twilio was employed. Twilio APIs [17] were used to send and manage outgoing SMS messages easily and without having to invest in actual SMS hardware infrastructure. Due to the expense of sending large volumes of messages for testing, a test message function was enabled that output mocked SMS messages to the logs for testing purposes.

D. *Testing*

In order to ensure that both functional and non-functional requirements were met, three sets of tests were run on the completed software packages: unit testing, acceptance testing, and performance testing. The results of these tests were then used to fix defects and to create a list of introduced defects for PSP defect tracking.

Unit tests were incorporated for each module and component to ensure that it was meeting all contract requirements and that its interfaces would behave as expected under varied conditions. Unit tests tested the ‘happy path’, ‘sad path’, and edge conditions for all modules.

Acceptance tests were used to ensure that all components were meeting the requirements they were designed to fulfill. They were a pre-created suite of tests based on the specified protocols for the MQTT, REST, and SMS interfaces that could be run on the completed system to determine if it would behave correctly at its external interfaces.

Finally, performance testing was used to ensure that non-functional requirements were being met. Two tests were run: A timing test from message delivery to console output (to simulate the SMS message) and an up-time test over a period of time to ensure that the service maintained high availability.

In order to measure the timing, a Python script was created that would pre-populate vehicles and contacts, then send MQTT messages at increasing rates and measure the time between its send and the time when a notification for that user appeared in the console. The rate of messages being posted would be increased from 1/sec, to 10/sec, to 100/sec, to 1000/sec and the delay measured to ensure that the timing never got over the 60 second mark. During each burst of messages, a single message would be tracked to see how long it took. This would be repeated multiple times for each message post rate.

A similar test was employed for uptime – a Python script was employed that would send constant HTTP requests to the system’s endpoints and listen for responses. A request would be sent every second, and any missed responses or timeouts noted as ‘downtime’.

VI. Results

Three metrics were used to determine whether the design and implementation met the requirements for this project: time from publish to notification, lost messages under load, and defect tracking against requirements and design in the final program. The following will present the result for the three cases:

A. *Message Routing Time*

Using a Python script to vary the frequency of MQTT messages being sent to the broker and to then measure the time elapsed until a message is sent. In order to test the time under high load, the number of messages sent per second was varied and the turn-around time tracked for a given message in each burst as shown in Figure 9.

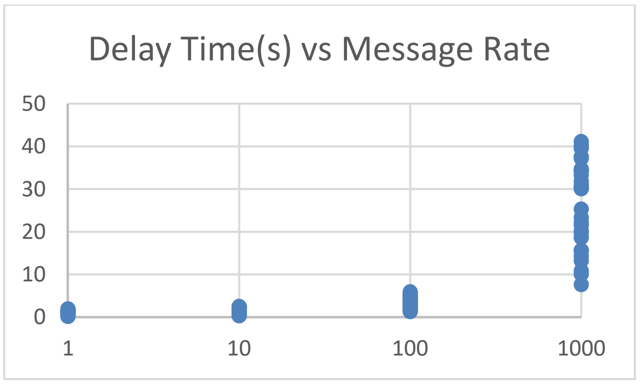

Figure 9. Delay Time vs Message Rate

We see that, for message rates in the 100/s range, the latency stays under 10 seconds, and with a top expected message time from Twilio’s services of 30 seconds, that leaves the system well within the 60 second requirement to ensure adequately timely notifications. However, as the 1000 messages/sec barrier is reached, we see latency times creeping up as high as 40 seconds – indicating that, for high usage scenarios, the expected routing time will not be met.

B. *Up-Time*

Using a different Python script to vary HTTP message frequency sent to the API endpoints, we measured the number of messages that failed to be handled under increasing loads. A

500 or timeout were considered failures as shown in Figures 10 and 11.

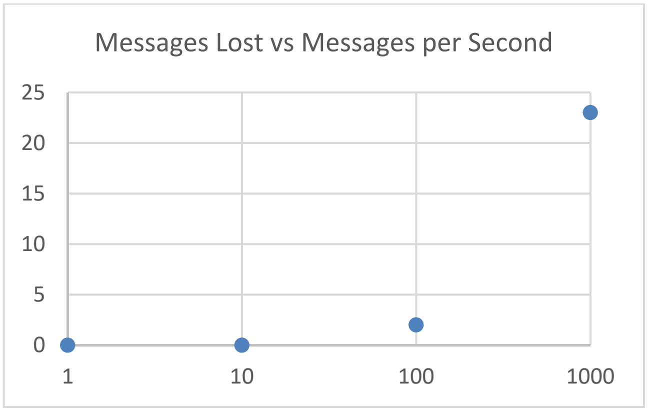

Figure 10. Total Messages Lost

As with timing, within the messages rates under 100 messages per second, the number of failure was under .001% - an acceptable bound. However, when the message rate gets to over 1000 messages per second we start to see a failure rate creep over the acceptable boundary.

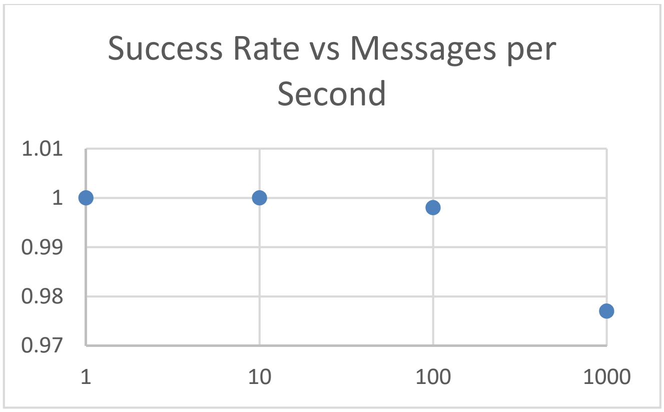

Figure 11. Success Rate

C. *Design and Requirements Defects*

Finally, defects were tracked against the design and requirements phases to ensure that the final product met the functional requirements for the system. Requirements defects were tracked against acceptance tests created during the requirements phases, and design defects were tracked against unit tests. Further, some tests were found to have defects that were discovered during the testing phase, which were also tracked as shown in Figure 12.

Defect Related To:	**Injected**	**Removed**	**Remaining**
Planning	0	0	0
Requirements	4	4	0
Design	15	15	0
Coding	24	24	0
Testing	3	3	0
Total	46	46	0

Figure 122. Defects

During initial testing, it was found that some errors related to requirements had been injected, and that a large number of coding and design related errors had also been created. Additionally, some small number of testing errors were discovered that were fixed before rerunning the testing suite. After the defects were fixed in the first pass, all tests passed and the software was considered to be meeting quality standards.

VII. Conclusions

Overall, this project was successful – all functional requirements were able to be met, and the non-functional requirements were achieved for a reasonable server load. The cloud based architecture provides a powerful tool for retrieving and processing sensor data, and because it is abstracted from the sensor themselves, provides an opportunity to update the messaging protocols for notifying emergency responders as new technologies and protocols become a requirement.

For performance, the system behaved well under what could be considered ‘normal’ loads, but saw unacceptable losses in performance as server loads began to creep into the high levels. To ensure that the system will be able to meet all requirements, it will be necessary to conduct further study as to what level of messaging should be expected from sensors in the wild – chatty devices will require redesign of the system. Second, load balancing may be required to ensure that all messages can be received under this high stress conditions, and to ensure that critical messages are routed appropriately.

References

[1] MQTT protocol specification (http://mqtt.org)
[2] SMS Protocol Specification (https://www.ietf.org/rfc/rfc5724.txt)
[3] N. M. Z. Hashim, H. H. Basri, A. Jaafar, M. Z. A. A. Aziz, A. Salleh and A. S. Ja’afar. "Child in car alarm system using various sensors." ARPN Journal of Engineering and Applied Sciences 9.9 (2014): 1653-1658.
[4] Ann S. ViksninsSuneel AroraJoan E. MorseKatharine A. Jackson Huebsch. "Warning system sensing child left behind in infant seat in vehicle." U.S. Patent No. 6,922,147. 26 Jul. 2005.
[5] Rossi, Marc A. "Warning system for detecting presence of a child in an infant seat." U.S. Patent No. 6,104,293. 15 Aug. 2000
[6] Guard A, Gallagher SS. “Heat related deaths to young children in parked cars: an analysis of 171 fatalities in the United States, 1995–2002”, Injury Prevention 2005; 11:33-37.
[7] Hantrakul, Kittikorn, Part Pramokchon, Paween Khoenkaw, Nasi Tantitharanukul, and Kitisak Osathanunkul. "Automatic faucet with changeable flow based on MQTT protocol." In Computer Science and

Engineering Conference (ICSEC), 2016 International, pp. 1-5. IEEE, 2016.

[8] Dhar, Prarna, and Poonam Gupta. "Intelligent parking Cloud services based on IoT using MQTT protocol." In Automatic Control and Dynamic Optimization Techniques (ICACDOT), International Conference on, pp. 30-34. IEEE, 2016.

[9] N. D. Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali, "Comparison of two lightweight protocols for smartphone-based sensing", 2013 IEEE 20th Symposium on Communications and Vehicular Technology in Benelux (SCVT), Namur, Belgium, 21Nov -2 Dec 2013.

[10] Jorge E. Luzuriaga, Miguel Perez, Pablo Boronat, Juan Carlos Cano, Carlos Calafate, Pietro Manzoni. "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks." Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE. IEEE, 2015.

[11] Tobias Häberle, Lambros Charissis, Christoph Fehling, Jens Nahm, Frank Leymann. "The connected car in the cloud: a platform for prototyping telematics services." IEEE Software32.6 (2015): 11-17.

[12] S. Bitam, A. Mellouk, "ITS-cloud: Cloud computing for intelligent transportation system", Proc. IEEE Global Commun. Conf., IEEE, pp. 2054-2059, 2012

[13] Gheorghe Panga, Sorin Zamfir, Titus Bălan, Ovidiu Popa. "IOT Diagnostics for Connected Cars." International Scientific Committee (2016): 287.

[14] REST Standard Specification https://www.w3.org/2001/sw/wiki/REST

[15] Spring Framework (https://www.spring.io)

[16] Mosquitto Project (http://mosquitto.org)

[17] Twilio Framework https://www.twilio.com