View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Virtual Commons - Bridgewater State University

Bridgewater State University
Virtual Commons - Bridgewater State University

Honors Program Theses and Projects | Undergraduate Honors Program

12-18-2013

Efficiency and Reliability of the Transit Data Lifecycle: A Study of Multimodal Migration, Storage, and Retrieval Techniques for Public Transit Data

Matthew Ahrens

Follow this and additional works at: http://vc.bridgew.edu/honors_proj

Part of the Databases and Information Systems Commons

Recommended Citation
Ahrens, Matthew. (2013). Efficiency and Reliability of the Transit Data Lifecycle: A Study of Multimodal Migration, Storage, and Retrieval Techniques for Public Transit Data. In *BSU Honors Program Theses and Projects.* Item 32. Available at: http://vc.bridgew.edu/honors_proj/32
Copyright © 2013 Matthew Ahrens

This item is available as part of Virtual Commons, the open-access institutional repository of Bridgewater State University, Bridgewater, Massachusetts.

Efficiency and Reliability of the Transit Data Lifecycle

Matthew Ahrens

Submitted in Partial Completion of the
Requirements for Interdisciplinary Honors in Mathematics and Computer Science

Bridgewater State University

December 18, 2013

Dr. Uma Shama, Thesis Director
Dr. Laura Gross, Committee Member
Dr. Larry Harman, Committee Member

BRIDGEWATER STATE UNIVERSITY

Efficiency and Reliability of the Transit Data Lifecycle

A study of multimodal migration, storage, and retrieval techniques for public transit data

Matthew Ahrens

12/4/2013

Acknowledgements

Special thanks are given to Dr. Uma Shama, my faculty mentor, and Larry Harman, Co-Director of the GeoLab for helping guide me on this and past projects. They have given me the opportunity to apply my knowledge on a living system and have let me see the direct effect my work has had in making people's lives better. I appreciate the communication and interaction between the GeoLab and Cape Cod Regional Transit Authority. Through being a primary resource and contact between their technology and database representative Aparna Sachidanand, Grant Manager Dennis Walsh, and Operations

representative Bob Lawson, I have been able to gain the real-world experience that only comes from working with other leaders in the field. I thank their patience, guidance, and the appreciation of my work they have shown throughout our projects. Lastly I would like to thank my Honors Coordinator, Dr. Laura Gross, as well as my co-workers in the GeoLab – current and graduated – for helping me continue to refine my work, code, and texts into the best that they can be. It is uncanny how many issues can be clarified and solved through short yet powerful discussion. I truly appreciate the advice, direction, and purpose they have given me.

If I have seen further it is by standing on the shoulders of Giants[1].

[1] Sir Issac Newton in a letter to his rival, Robert Hooke. Feb. 5, 1676.

Table of Contents

Contents

Acknowledgements 2
Table of Contents 4
Index of Tables and Figures 2
Introduction 3
Background 3
Thesis 4
Overview of Sections 5
Specifications and Ideology 6
Research Questions 6
Purpose 6
Methodology 7
Results 8
Static scheduling and informational data 9
Dynamic scheduling and informational data 10
Real-time data 10
RTA specific data structures 11
Discussion 11
Conclusion 12
Efficiency of Data Migration Techniques and Concurrency Models 14
Research Questions 14
Purpose 14
Methodology 15
BusLocator – a study 15
Next Bus ETA Helper 17
GTFS and NextBus 18
Results 19
Pull design pattern and improving transmission quality 19
Discussion 24
Security and the closed pipe vs the open pipe 24

Conclusion 26

Design considerations when moving forward and making GLaaS 26

GLaaS Model and Verification of Database Design 27

Research Questions 27

Purpose 27

Methodology 28

Initial Design – triggers and the trickle down structure 28

Subsequent Design – Batch Query 32

Results 33

Discussion 34

Criteria for success 34

Conclusion 34

GLaaS API and Validation of Client-Service Architecture 36

Research Questions 36

Purpose 36

Methodology 36

Tool choices 37

The business logic and application levels 41

Authentication and security 42

Results 42

ServiceStack, SignalR, and a multimodal approach to data services 42

Discussion 44

REST vs. SOAP 44

CLR Triggers vs. SQL broker 45

Conclusion 46

Final Statements 47

Index 48

Bibliography 49

Index of Tables and Figures

Figure 1GTFS definition for the block data field. A dependent of the trip structure, from a user oriented perspective. 7
Figure 2 APTA's TCIP definition for a block. Similar, but from the RTA perspective. 7
Figure 3 A model showing that there can be independence of production and consumption rate at either side of the data sink. 8
Figure 4 A composed encapsulation of the superset, ETA, being composed of simpler transit data structures and primitives. 11
Figure 5 A graphical depiction of a simplified BusLocator 16
Figure 6 The data flow of ETA data collected and analyzed by NextBus ETA Helper 17
Figure 7 Application example of on-demand, pull design pattern for transit data. 20
Figure 8 Real-time, mapping application that uses the pull design pattern. It displays the most recent AVL data for CCRTA vehicles at 5 second pull intervals. 20
Figure 9 Summary of Exp. Mov. Avg. Algorithm simulation on 10 vehicles. 23
Figure 10 A point chart of pulled AVL points from both algorithms between AVL id's 100 and 400 23
Figure 11 An example of an application level query in a c/java/c# like syntax that can lead to SQL injection. 30
Figure 12 An example of a WCF data contract (XSD) that is publicized to the client. 38
Figure 13 The URI route binding and auto generated metadata documentation of a sample ServiceStack project. This API services the Feature for AVL. 39
Figure 14 The human readable metadata for a ServiceStack API with compatibility for SOAP and WCF client consumers. 40
Figure 15 Overridden Route definitions for this operation. Defined by the request DTO and not by remote procedure calls gives flexibility over verb usage, URI chaining, and embedding the data in the URI bypassing the need for a response object in some cases. 41
Figure 16 Request DTO definition as a lightweight, serializable JSON file. 41
Figure 17 Response DTO definition as a lightweight, serializable JSON file. 41

Introduction

Background

Student research assistants at the GeoGraphics Laboratory – a GIS research lab at Bridgewater State University – have to support many systems and act in a variety of roles. The GeoLab fulfills the transit data needs of many RTAs[2] through maintenance of existing systems; mediation on proof of concept, product and project decisions; consulting in areas of technical expertise; and developing new applications for existing and prospective grant projects. Student researchers are expected to act as experts in these areas to best facilitate the RTAs. The skills required range from the political and authoritative, to the communicational and instructional, to the applied and very technical. Staff turnover rate for the GeoLab is very high compared to non-academic institutions of similar purpose as student researchers typically do not begin until their junior year of undergraduate study or until they are graduate students. Knowledge of systems is inherited in the form of inline comments; version controlled log entries; or interpreted from existing code and project presentations. These considerations of programmer time and the student developer's experience at the GeoLab greatly influenced the purpose of this project.

In documenting this project, future GeoLab researchers should be able to understand the decisions made in designing this new system. In previous projects such as the GeoLabVirtualMaps[3] website and the BusLocator Windows/WCF[4] service, intention and use had to be interpreted and assumed. This project includes a protocol that can be followed to incorporate recent best practices. Hyper-text documents created can be garnered for understanding and the decision structure behind major aspects of the projects. Lastly, the methodologies utilized by this project are designed to be replicated in future development. This project has been made realizing it is not complete. It will continue to be maintained and expanded upon – and ultimately replaced – as new technology becomes available and web communication improves. This project has been made in such a way to best facilitate that purpose and act as a template, protocol, and guideline for migrating transit data.

This project has evolved greatly from its initial conceptualization. The first purpose of this project was to provide efficiency and optimization analysis on data collected from Cape Cod Regional Transit Authority (CCRTA)[5] Vehicles during the spring of 2013. Key areas of interest were AVL[6] data collected from the vehicles' GPS MDTs[7] and customer consumed Estimated Time of Arrival data provided by the third-party, service provider NextBus. The purpose was to determine the quality and accuracy of data under different parameters such as date, location, and time and along the different stages of the data's lifecycle. It became rapidly apparent, however, that the data collection mechanism was not expandable to facilitate analysis functionality without interfering with its original purpose of moving and processing data from source to client. Over the summer of 2013 the data collection mechanism was to be replicated instead of extended. The purpose was to avoid the previous term's

---

[2] Regional Transit Authority

[3] Web application used to display Vehicle AVL Data on a map updated in 5 second intervals. Hosted at http://www.geolabvirtualmaps.com

[4] *Windows Communication Foundation*, a runtime and set of APIs for developing request-response based web services.

[5] Cape Cod Regional Transit Authority. Provides Fixed route bus, Paratransit bus, and Rail services to the Cape Cod region.

[6] *Automatic Vehicle Location*. System used for determining and transmitting the geographic location of a vehicle.

[7] *Mobile Data Terminal*. A programmable terminal installed inside vehicles for wireless communication with an RTA.

issues of dependency while reconstructing the same type of program for ease of use. A program, called Next Bus ETA Helper, was created to resemble the concurrency model of BusLocator. It utilized .NET timers to make pull requests at regular intervals. Each request occupied its own thread and could work simultaneously with other requests. It had a short data lifecycle, pulling aggregate ETA[8] Data provided from NextBus' public, XML Webservices in two minute intervals. This data would contain tags about vehicle direction, destination, and route and trip metadata. The request would be parsed by the windows service and stored in a database structure similar to the AVL structure created for BusLocator and GeoLabVirtualMaps. The primary goal of this program was to pull a data source for analytics. The predictions contained in the ETA data feed was to be compared to the AVL data used to calculate it. Vehicle arrivals would be noted in the system and accuracy was to be correlated to distance, traffic, time of day, weather, or other parameters.

This process encountered its own issues and made clear the distinct areas of transit data migration that needed to be addressed:

1. Concepts and definitions of transit data were unclear between different authorities
2. The general tools used in the current data collection system, while complete and functional for their initial purpose, could be greatly improved
3. Extendibility and replication were unachievable, only maintaining original purpose was realistic
4. Any replacement of the existing system would need to be integrated in parallel and include training and documentation materials

Consideration for future work was integral in creating these resulting works during the fall of 2013. While it was unrealistic to assume that a complete refactoring of the existing system was possible in the remaining amount of time, it is possible to provide the framework for such a system. The works of this project were done iteratively as outlined by the Agile Software Development philosophy. Documentation for future student researchers was created with the purpose of future collaboration. Protocols and working implementations of how to create/replicate, use/consume, and extend each primary feature of the project was made and accompanied by simple tests to validate their results. If this framework is followed, future researchers can learn from the work of previous projects in the lab and not have to recreate and relearn the entirety of existing projects in order to benefit from their utility.

Thesis

Designing, implementing, and analyzing transit data services reveals four key areas of study: (1) There is a need for clear, reliable definitions of transit data structures that satisfy the various authorities, protocols, and specification; (2) Current protocols and specifications dictate the computational tools and techniques that must be used with their data definitions and the two should be uncoupled and analyzed separately; (3) Choosing to make a context-less data model that is separate from the business logic of its use enables best practices and significantly improves quality of data storage and retrieval in transit systems; (4) A multimodal approach to API design can make a flexible, secure, and purposeful client interface to the transit data and eliminates any necessary knowledge of how the data is stored or retrieved.

---

[8] *Estimated Time of Arrival*. Time interval between the expected arrival of a vehicle at a particular location represented as a data point.

Overview of Sections

The four sections of this project were made so that future GeoLab researchers aiming to continue this framework need not know every part to make substantial and purposeful progress. On the contrary, it should be that they are able to focus on a particular aspect and provide expert opinion on that matter while simply utilizing the other, completed parts of this framework. If they are able to do so, then the original issues encountered when beginning this project have been addressed and ameliorated.

The order of the sections is from design to implementation. The first section is descriptive of the authorities and appeals of this project, the different entities that have influenced this project, and the definitions, terms, and concepts that will aid future development on the framework. Theoretical computer science and best practices of big data migration is contained in the second section. This section will aid designers of new systems in understanding which techniques, design patterns, and program architectures they should use when replicating the purpose of this framework. The third section is descriptive of the database and model schema that is implemented in this framework. It contains the relationship between concrete Database Objects such as tables and queries, and also the structures the framework invented for best use. This section is important for researchers looking to extend or replicate the capacity of the framework for new transit data structures, new GeoLab data types, and other unforeseen uses. The last section is the implementation and use case of the client facing API. It shows how the data can be stored and retrieved from the consumer or providers point of view. It describes specific technology implementations, such as ServiceStack and SignalR, and shows how these implementations improve round-trip time and data load on the server. Researchers looking to implement security mechanisms for specific data, add features or support for a particular service, or consume data for their particular research project can consider this section for best practices.

The specific implementation of the framework described in the last two sections has been named *GLaaS* – **G**eoGraphics **L**aboratory **a**s **a** **S**ervice. The name is a play on the SOA[9] programming paradigm SaaS – Software as a Service. Since the GeoLab provides specifically tailored data services – data feeds, text file databases, Web Applications – and not redistributable, contextless software, the name is befitting of the intention of the framework.

[9] *Service-Oriented Architecture*. The design pattern of providing functionality as services to other applications independent of software product or use.

Specifications and Ideology

Research Questions

How can the different transit data protocols be described to compromise between conflicting definitions and structures? Is there a compromise that can be reached that is still purposeful and clear?

Purpose

Becoming familiar with the various data structures and relationships between data definitions is the initial step in understanding the problems and concepts of public transit data management. Public transit data structures will be defined and explained by both their structure and applied use. The different protocols and specifications governing transit data integrity and agreement will be compared and separated from their specific implementations. In pursuing these definitions, the conflicting needs of different RTAs were considered. It is difficult to speak outside the specific scope of data this project was privy to, and many RTAs disagree on specific terms. The highest representative authority was given priority in these cases of disagreement.

The need for data definitions within the scope of this project originally resulted from modifying an AVL service originally used to simply transport AVL data from CCRTA to the GeoLab. The intention was to accommodate services provided by NextBus by sending a copy of the data to their service endpoint as well. While all parties involved in the discussion of this feed had a complete understanding of transit data concepts, there was disagreement over the definition of the data structure *block.* It was unclear what a block was, how it was calculated, and what it should represent. The issue manifested from translating the practical definition and alternate terms used in the RTA, to the structure labeling of the associated data in the proprietary AVL database, and to the ETA related definition provided by NextBus' specification. The RTA definitions came from how the data was used in practical business application, and the database definitions came from the medium or location of where the data was stored. When comparing the GeoLab and CCRTAs understood definitions of transit data structures to NextBus, it became apparent that NextBus had an implementation-oriented point of view of each definition. Each data structure was defined by how it was going to contribute to calculating the value of the ETA between the vehicle and each stop location. In order to find a compromise, GTFS[10] – the static scheduling standard that both CCRTA and NextBus were familiar with – definitions of types were compared. GTFS has a user-oriented perspective of each definition. The definition of an object is determined by how the end user will interpret that data in their map-based application. Moreover, the definitions were tightly-coupled to the text-based database model their GTFS was defined under. The highest two authorities on the subject were considered: APTA's TCIP[11] and ESRI's GIS[12] definitions. TCIP is the data definition standard of one of the most highly recognized transit entities known. ESRI is the de facto standard from the private sector recognized by the OpenGeo Consortium through their creation of GIS protocols and file formats such as GML. TCIP was written from the use of the RTA, and assumes the proprietary or in house definition of most data types. In some instances, such as with the definition of block, alternatives are given and definitions are more like descriptions. The purpose of the TCIP was to provide a structure for inter-operable data communication between RTAs and departments, but not to actually define the data itself. The ESRI definitions are the most generic and applicable of the definitions,

[10] General Transit Feed Specification, a data structure for static, RTA scheduling information such as that which is printed on paper transit schedules.
[11] American Public Transportation Association's Transit Communication Interface Profiles. A data definition and initiative for inter-RTA communication and interoperable data.
[12] Supplier of ArcGIS and other GIS applications, the national standard recognized by the OpenGeo Consort. to provide standard Geographic Information System definitions and file protocols.

but they are not tailored to transit. This leaves them as a suitable reference for understanding, but not in defining a strict protocol.

block_id	Optional	The **block_id** field identifies the block to which the trip belongs. A block consists of two or more sequential trips made using the same vehicle, where a passenger can transfer from one trip to the next just by staying in the vehicle. The **block_id** must be referenced by two or more trips in trips.txt.

Figure 1GTFS definition for the block data field. A dependent of the trip structure, from a user oriented perspective.

	ratio of bits received in error to bits sent.
Block	A vehicle work assignment.

Figure 2 APTA's TCIP definition for a block. Similar, but from the RTA perspective.

Methodology

Transit data structures can be broken down into three categories: Static and informational data, dynamic scheduling and informational data, and real-time data. These categories are made based on their frequency of change. The aforementioned authorities would categorize the different structures into their frequency of consumption. Through the desire to repurpose data at variable rates of consumption, it is apparent that those definitions are inadequate. Therefore the frequency of change refers to the rate and variability which the data in the data structure is produced. For example, AVL data produced from a vehicle changing location occurs at a much faster rate than scheduling data that changes once per season. What constitutes when data moves from one category to the other can be disputed by different providers. For the purpose of clarity the following definitions are made. Static data is the least frequent to change and is only changed at predetermined intervals or known times. Dynamic data is infrequently changed, but it is not known beforehand when or why the change occurs. Real-time data is expected to change frequently at variable rate intervals which cannot be predicted. The categories not only help decide which data structures can be classified together, but also how best to compose them in order to have a clear, definitive understanding and description of a data model.

By having these distinct categories, transit data structures are independent of how they are stored and used. They are still dependent on how the data is made. This is a fundamental step in data reuse, and can alleviate some of the dependency problems posed so far. This also helps solve disputes about specific data structures between RTAs. The functions that an RTA provides – e.g. vehicle services, scheduling services, inter-RTA operations, etc. – typically are the same in action and concept, even if details and management technology or protocols vary. By tailoring this project to the fundamentals of transit data in the abstract, we are temporarily ignoring implementation level interoperability problems to inspire solutions that will work beyond the current technological standards.

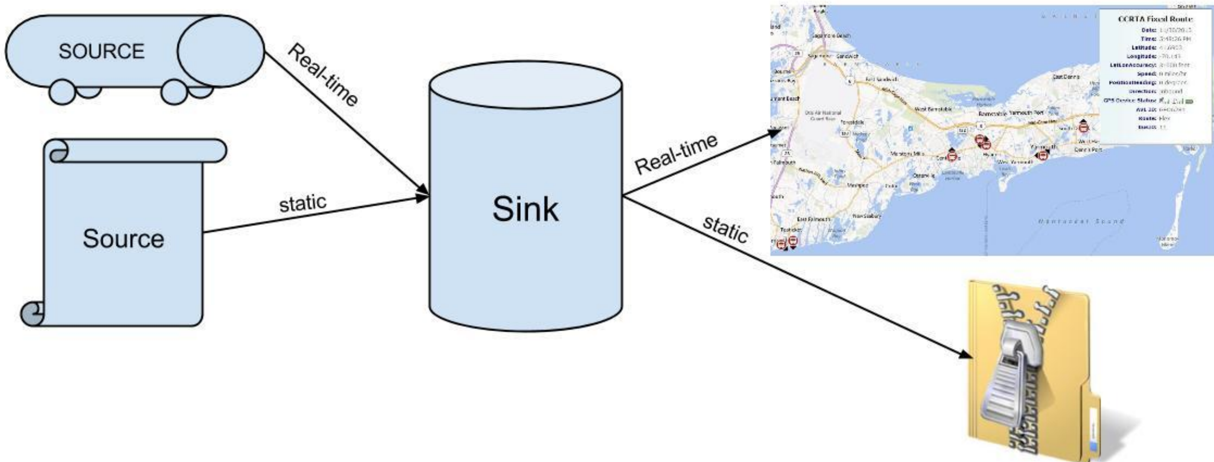

Figure 3 A model showing that there can be independence of production and consumption rate at either side of the data sink.

A set theoretical approach to defining transit data structures was chosen to best embody these ideals. By defining transit data structures as sets of concrete, implementation-free properties, they satisfy the conditions of reuse. The sought after object-oriented programming principles of encapsulation[13] and extendibility[14] can be achieved by composing large transit data structure out of multiple simpler ones. Invoking these two principles limits the overhead and contextualized metadata needed to package, store, and retrieve data for multiple uses. Lastly, ambiguity is removed by defining the properties by how the data structure is made. Some assumed knowledge has to be made, but any non-transit specific data used in these definitions can come from GIS definable terms or algebraic mathematics.

Results

The **primitive types** assumed by this set theoretical definition of public transit data are the following: *geolocation, datetime, and unique id.* A **geolocation** is any coordinate system data type that provides an exact geographical location. A latitude and longitude numerical coordinate are typically used. Geolocations are unique meaning that two geolocations refer to the same coordinate if and only if they are equal. Datetime in transit data is used for the ordering of events. Two transit data structures are simultaneous if and only if their datetimes are equal. Datetimes are typically stored numerically—e.g. 20130712 instead of July 12th, 2013 – in order to take advantage of comparative operations such as greater than or less than. A **datetime** that is greater than another is further in the future. While many different standards exist for datetime, the easiest one found and utilized by this project is akin to the T-SQL[15] datetime regular expression: YYYYMMDD HHMMSS.F. It can be read as: year, month, day, hour, minute, second, fraction of a second. By giving the data structure this organization, it can be numerically ordered easily by any data management system. By providing a fraction of a second, uniqueness can be determined to any precision possible in implementation. The most frequently used storage oriented competitor to this data ordering has been the combination of days from Jan 1, 1970 and seconds (or

[13] Programming principle of defining an object by its characteristic properties and functionality. E.g. A Student object encapsulates a name, a gpa, a student id number, etc.
[14] Programming principle of defining an object by inheriting all of the characteristics of another object, and then adding its own. E.g. all students are people, so students extend the characteristics of people.
[15] Structured Query Language. Programming language used in relational databases to store, access, retrieve information in a table-like structure.

milliseconds) from midnight for date and time, respectively. The benefit gained from this is a performance increase to typical storage mechanisms and ordering since both numbers are stored as integers. There are two main costs to this. The first is that it is not visibly human readable and must be computed to an easily comprehended value to be read. While order can be gained from comparing the numerical values, a sense of hierarchal grouping cannot. For example, calculation must be performed in order to determine what values are within the same month, day, or year. The second is that the granularity is set by the storage mechanism. By having an integer for the time portion, it is not allowed to have fractions of that time slice, thereby not letting the definition of the datetime be free of the context of its storage mechanism. This results in unnecessary loss of precision. The **unique id** of a data structure is abstract and can be best summarized as a rule. It is the statement that two transit data types are equal (and can be given the same id) if and only if their corresponding properties are equal. While almost unnecessary to write, this primitive type is used quite often when organizing data structures in collections and sets. Therefore, it should be stated what constitutes uniqueness of identity, even if it is begging the set theoretical definition of equality. Given these fundamentals, the transit specific data structures can be explored. Metadata or other contextual niceties may exist for these data types, but these are the context free, concrete definitions that this project sought.

Static scheduling and informational data

Vehicle

The simplest transit data structure is the vehicle. A vehicle is composed of one unique id property, a vehicle id. A vehicle id must be complex enough to distinguish it from all other vehicles. Many RTAs provide unique numerical ids to their vehicles. But two RTAs typically do not guarantee uniqueness between themselves. Typical use case is a unique textual prefix to denote the RTA and/or service, and a unique numerical id pertaining to a particular vehicle.

Stop

A stop is a grouping of a unique id property, a stop id, and a geolocation. A stop is simply a named place, or a point of reference. Ambiguity and violations of uniqueness are not typically encountered with stops as they are a RTA contained data structure. Any dispute can be resolved by comparing the geolocation in the case of redundant identification Therefore, in the rare case of scrutiny, a stop can be the unique pairing of stop id to geolocation.

Trip

A trip is an ordered collection of stops. Its composition is defined as the ordered collection of stops that a vehicle will traverse in that particular order. The order can be determined from a linked-list structure; a vehicle must visit stop at index 1 before stop at index 2. The order can also be determined from a 2Dimensional array-like structure; a vehicle must visit the stop at index i at time t_i and $t_i < t_{i+1}$. Therefore, a trip's set definition can optionally include a corresponding collection of datetimes, but in either case stops in a trip correspond to a chronological order. If the trip does not include a collection of datetimes, it should be defined what datetime a trip begins, and at what frequency it arrives at each stop. From this information, the exact datetime a vehicle should be at a given stop can be calculated. Lastly, a vehicle must have adequate time and resource to get from one stop to the next in physical space for a trip to be valid.

Route

A route is a stop-oriented, ordered collection of trips. Each trip has a similar collection of stops. It is an organizational structure for practical use and not computational or predictive use; the route is a way of organizing trips by likeness and time.

Shape

A shape is an ordered collection of geolocations that denote a path of traversal – sometimes called a polyline – through a geographic region. Shapes are commonly used to outline directions or other paths a vehicle will take when traversing a trip, but can be used to outline any connection between geolocations or stops.

Dynamic scheduling and informational data

Block

A block is a vehicle-oriented, ordered collection of trips. A block is created as the collection of trips a vehicle will take in sequential order. A unique consideration must be made with blocks. If a vehicle is on a particular block at a particular time, then its position in the trip (what stop it is heading to and/or arriving from) is calculable. Therefore, a block cannot contain trips whose stop sequences have overlapping times, and a vehicle must have adequate time and resource to get from one trip to another.

Status and conditions

A status or condition is the association of any of the static data types with metadata or context. Typically a mechanism for providing current information such as notifications, weather and safety alerts, detour and delay information, etc. to existing services that do not change. The particular reason that this date structure is dynamic and not real-time is because of its intended creation and lifecycle. This data is used to add informational context by the data collection mechanism. It is kept separate for this reason and can be seen as metadata. Status and conditions are typically persistent and are unexpected to change frequently. They are determined not on regular intervals, but rather by the start and stop of discrete, unplanned events. While this definition may be intentionally abstract, it is much more understandable than definitions that limit what a status or condition can be. By making it the composition of any previous data type and some textual or numerically identified notification, any adopter of this definition can fully identify data of this type while still having the most freedom of implementation.

Real-time data

AVL

AVL is the abbreviation of Automatic Vehicle Location. An AVL data structure – sometimes called an AVL data point – is the set containing a vehicle, a geolocation, and a datetime. When reduced to its minimalistic definition, the name clarifies the elements of the structure the best. The creation of AVL points the name describes must be automatic. Therefore, it must be set to an autonomous process. Most autonomous processes are based on time interval or have some knowledge of the time at which they occur. That automatic process is taking the location, which can be stored in a geolocation, of a particular and unique vehicle, which we can represent with the vehicle data structure.

ETA

ETA is the abbreviation of Estimated Time of Arrival. An ETA data structure is widely disputed between systems. These systems define ETA as what is most convenient for its intended use. In Next Bus applications, it is as simple as a vehicle and a time interval. In Trapeze's database structure, there are many metadata contexts that are indistinguishable from the required minimum definition. By breaking down ETA by how it is created, it becomes apparent what is the minimum to satisfy all required uses.

An ETA can be defined as the encapsulation of an AVL point, a stop, and a datetime. This satisfies how the ETA is created considering: an ETA data point is created when a particular vehicle at a particular geolocation and datetime (AVL) will arrive at a stop at an estimated (calculated) datetime. It

can be determined if two ETAs are unique, if an ETA is accurate, if ETAs are similar, etc. from this definition.

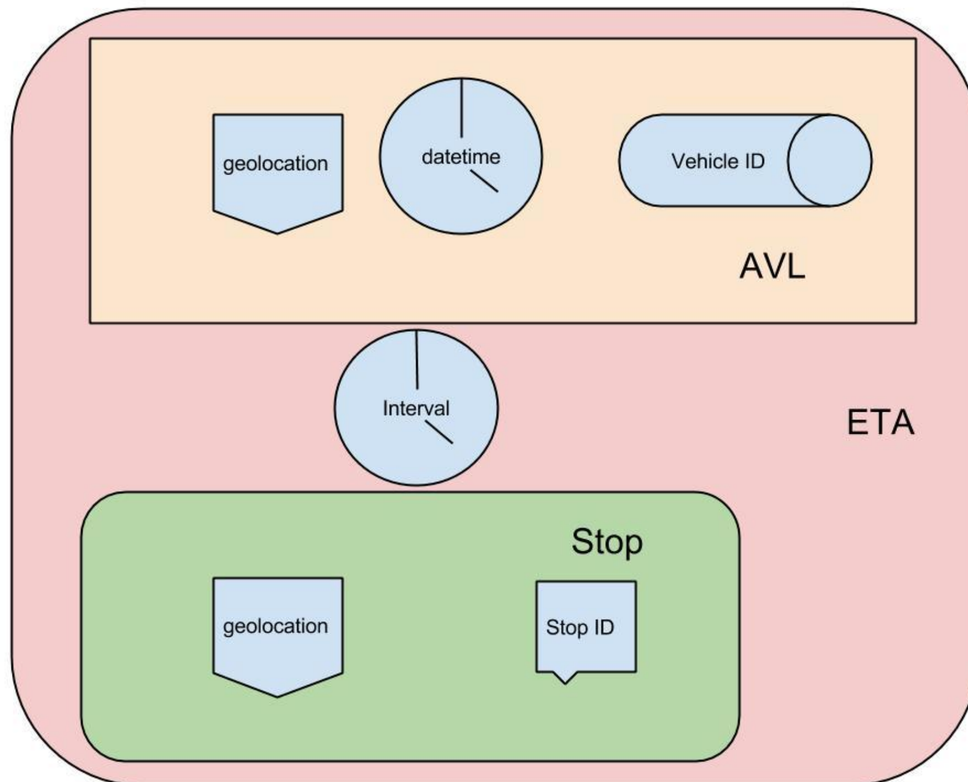

Figure 4 A composed encapsulation of the superset, ETA, being composed of simpler transit data structures and primitives.

RTA specific data structures

There are many infrequently used data structures that exceed the scope of this project. These include the data structures for fare payment and monetary data, personnel and customer information, and the useful, application specific inheritance of the aforementioned types. It is expected, though, that by continuing this set theoretical definition, these types can be derived from the existing ones presented here. Or new, unrelated ones can be defined in a similar format.

Discussion

A major concern of the three categories is what constitutes real time. ESRI, a leader in GIS technology and standards defines real-time data as, “Data that is displayed immediately, as it is collected. Real-time data is often used for navigation or tracking.”[16] This definition, while accurate, has led to confusion when defining transit data structures. For example, consider a typical paper bus schedule for any transit authority. If the transit authority utilizes a technology that displays their bus schedule as soon as it is written and committed on a website, then the bus schedule meets this definition for real-time data. It is understood that this is not what is meant by real-time data, but the definition allows it to fit. GTFS provides a closer ideal to an approved definition of their specific subset of real-time data from the user-end perspective. “GTFS-realtime is a feed specification that allows public transportation agencies to provide real-time updates about their fleet to application developers.”[17] This

[16] ESRI GIS Dictionary, http://support.esri.com/en/knowledgebase/GISDictionary/term/real-time%20data
[17] GTFS-Realtime https://developers.google.com/transit/gtfs-realtime/

definition addresses the category as a superset of data structures. The specific structure in this case is a *feed*[18]. The GTFS's inclusion and assuming the understood use of the term *real-time* in their definition is not intended to this issue of dependable data integrity, thereby not giving any criteria clarity. The definition concluded to in this section for real-time satisfies the authoritative criteria, while also being precise.

In making the specific choices of having an implementation free, context free data definition, there exists cost. More work has to be done on the part of the implementer of these definitions to decide which delivery and storage mechanism they wish to use when interacting with the data. It is possible that adding a metadata tag onto one of these structures might make it erroneously similar to another structure and cause confusion. The primary idea to take way from this work is that the data to medium should be a one-way function. These data types should be implementable in any environment or technology that will have them, but the environment should not dictate a definition.

Data structure specific discussions

While trivial in definition, AVL is one of the most important real-time data structures and the most widely used in customer-facing GIS applications. It is the building block for many of the other real-time structures and is the primary example of this project's data collection evaluations. Understanding AVL and what it consists of (and what it does not consist of) is crucial to success in real-time GIS services. It can be paired with many other data services such as ridership and pay history, boarding and alighting, arrival and proximity, etc. to make aggregated data types not described here. The retrieval, storage, and utilization of AVL data is the primary example discussed in the following three sections of this paper. The intent is that by having this fundamental strategy on how to dependably store and retrieve the most representative of real-time data, any future GeoLab project can be made easier.

An ETA data point is the most misunderstood of real-time data. It is typically defined in its end purpose. The typical end purpose of an ETA point is the numerical value of time as an interval. The rest of the data is contextualized in how it is retrieved or calculated. An end user application, such as a map or PDA display, only cares about the numerical interval given a particular vehicle and stop at a time – right now. By calculating that number for its immediate use, being displayed, and excluding reference of the defining parts – the AVL and stop information – the ETA point loses any reusable application for historical information (ETA performance algorithms, Bayesian inference and statistical models, etc.) or any use for quality assurance and verification (confirm the calculation is reasonable or correct).

Conclusion

It was seen that a set theoretical definition of transit data structures provides the minimum requirements that make the data unique, assessable, and clear. While implementations of the data will vary, and extensions of the data will add on metadata, structures, and technology specific contexts, the integrity of this data can provide a stable foundation on which to build transit applications. Future researchers using this document as their basis for understanding the building blocks used by the GeoLab to provide services for transit should be empowered to understand and separate the data from the tools they will work with.

[18] A feed is a collection of data structures that is appended to over time. It acts as a fixed length snap shot that a consumer can periodically check for updates to a resource or history.

Purposeful resources can be built upon this definitive structure. Storage mechanisms can be made interoperable[19] if they are designed in a way that agrees to maintain the integrity of the definitions. The three categories of data structures, defined by the frequency of data creation, have been decoupled from purposed implementations (applications and use) and transmission mechanisms (technologies). It is at this stage that the pairings of data and technologies can be scrutinized and assessed. The technology, simply the tool, can be best compared to the job or use. Even more importantly, a combination of tools that best facilitates the end goal can be composed in order to gain the most utility.

[19] The act of transmuting data from one system to another, while losing a minimal amount of information in the conversion. Standards, protocols, and specifications exist to make interoperability easier since no information should be lost due to no conversion being needed.

Efficiency of Data Migration Techniques and Concurrency Models

Research Questions

What technologies, techniques, or models most efficiently and reliably move transit data from producer to consumer? Which of those best embody the concepts of reuse, extendibility, and reusability? Which ones are resistant to need modification and internal maintenance?

Purpose

The current Transit Data service that the GeoGraphics Lab maintains and utilizes for most of its real-time data needs is BusLocator. BusLocator is the inspiration for this project. Maintaining it, extending it, replicating it, and now replacing it were the stages of development this project iterated through. BusLocator and all of the AVL data work and production that the graduated student researchers built upon BusLocator is not lost in this change. The decisions they made were the stepping stones. The purpose of this section of the project is to provide a comparison and contrast of the techniques utilized in BusLocator among other techniques used commonly in storing and retrieving transit data given the definitions and structures in the previous section. As technology becomes sufficiently sophisticated, it is up to the engineers and researchers that use that technology not to simply maintain or produce for the present, but to automate and innovate for the future. The results of this section outline the choices and decision structures made when implementing the GLaaS framework discussed in the last two sections.

This section is concerned with finding the technologies that fit two distinct parts of the software or service development lifecycle: Verification and Validation. *Verification* is the act of building technology that fit their written specifications and goals. *Validation* is the act of building technology that best fits its use or application. These two measures of software development effectiveness have been used for assessing high-volume data simulations and other service oriented software[20]. The services the GeoLab provides to RTAs for their data services have grown to fall under the same categories, and the problems have grown to be of a similar magnitude. The research products and proof-of-concept services developed by the GeoLab under their various grants have been exceptionally good at verification. Due to high turnover rate, evolving grants, and the small team nature of daily lab tasks, validation has not been addressed. It has not been until this iteration of student researchers that time could be spend on assessing the quality and encompassing themes of the applications developed. By having this period of refactoring and reflection, the data migration services that the GeoLab provides can have a significant increase in success of validation. As team documentation tools have improved in the lab, it is apparent that the majority of time is spent on debugging and maintenance. As new grants have requested the Lab to recreate past work, it has been apparent that many of the proof-of-concept projects were not developed with the intent of extendibility. Lastly, as research became reflective during a time that was in between new venture projects, the need to refactor and reassess the current state of GeoLab services was apparent.

---

[20] Engelbrecht, R., Rector, A., Moser W., *"Verification and Validation" Assessment and Evaluation of Information Technologies in Medicine*

This section will outline the method which was used to survey tools for facilitating common GeoLab tasks when developing new solutions. These solutions can be considered as student researchers are currently automating their manual tasks when maintaining AVL mechanisms, generating GTFS databases, or tackling new grant objectives. The desired outcome is a cohesive lab culture that: minimizes having to learn or re-learn lab technologies and solutions; create a design architecture that fights anti-patterns inherit in specific tools; and lends itself to purposeful documentation and structure that is best fit to the task.

Methodology

The best practices discovered during this section of the project were tested iteratively. Due to semester iteration of project goals through grants, credit commitments, presentations, the lifecycle and purpose of these projects fit best in the agile software development lifecycle[21]. This arrangement was not intentional, but once realized the benefit of it was garnered for its full potential. Documentation was created concurrently with iterations of the projects, tests, and use. Any time spent on design and documentation was purposeful and could directly translate into tangible work or products. Implementation of the current project was in the context of connecting previous and future projects so that definitions, protocols, and algorithms applied in one project could be easily reused in the next. Dividing tasks and assigning projects between researchers at the lab was done as to best balance maintenance, current projects, and future prospects. The following studies show the projects visited and the practices discovered over the course of this projects lifecycle.

BusLocator – a study

BusLocator is a Microsoft c# solution that consists of two parts: a data producer and consumer. The producer program is made from a c# windows service project that is based on timer-event concurrency using callback functions. Event based concurrency is a standard for performing an action, subroutine, or any task based on a condition or interrupt. BusLocator was designed to associate a timer at a constant time interval with each category of data. Categories originally included AVL, ETA, and route data for fixed route vehicles. The benefit of this concurrency pattern is that it is simple to implement and understand. To replicate this pattern is easy as well. To add a new feature or associate with a new category of data, the developer simply adds a new timer. BusLocator is hard to modify however, as maintaining one aspect of the system service requires interrupting every feature. Also, unhandled errors and debugging in one feature can halt the entire service. Service design best practices, exception handling, and replicating and extension over modification did aid in preventing these issues, but other problems arose from its concurrency model.

Upon event activation, a method is called to handle the entire operation of data transfer from database, T-SQL and SQL Server in this implementation, to the consumer over the network. It faces two network connections on either side of the transfer. It occurs at a regular interval, which does not account for fluctuations in the data load, and holds the success of the entire batch pull of data against the success of each of its parts. Specifically it is in a design patter, that while typical and successful, does not lend itself to data assurance and safety. Variations in network load and traffic, connection states and properties, database efficiency, load, and data set side, among other unforeseen interruptions can cause the drop of an entire batch.

[21] A software development methodology based on iterative and incremental development where requirements, purpose, and solutions naturally occur through collaboration and exploratory research.

BusLocator was originally created with the intended purpose of real-time data consumption. This is to be distinguished from the real-time data structures from before. BusLocator was designed with the purpose of providing data in a reasonable time after it was created for use and consumption. This led itself to a common big data[22] problem that comes before analytics can even be performed. The problem was that the data did not maintain the integrity of the underlying real-time structure. The data was regularly, but not reliably transferred to the consumer making it not fault tolerant. This was not a problem for average, real-time use; a public transit patron does not notice a one to two minute gap between AVL points on a mapping application or notification system. This was a large problem for analytics and quality assessment, however. In trying to judge the integrity of NextBus ETA data, as was an original intent of studying the data gained from CCRTA, the point of reference defined for ETA, AVL, was inconsistent. The granularity of the collection mechanism was dependent on the technology of the transfer mechanism and not on the capability of the source and the data.

The consumer counterpart of the BusLocator solution is a c# WCF service that acted as SOAP[23] endpoint. That is, an open endpoint for the windows service or other AVL, ETA, and Route service providers to POST – send – data to. While WCF has been currently criticized for being an anti-pattern which actively fights the protocol – HTTP – on which it's built, it is still a utilized and dependable technology. This service endpoint provides a scalable, concurrent, and reliable mechanism for accepting data and safely storing it in the database. Independent requests do not conflict with each other. Failure of one entry does not interfere with another. The actions are performed on demand, and minimal delay is produced between initialization and completion. Maintenance of this system has been incredibly simple due to its organization.

Complications arose when trying to modify the structure to handle changing expectations and features from grant objectives. When trying to add functionality to store the data from a new feature, it became apparent that the data model in the Lab's local database was completely dependent on the business logic of its use. Business logic is the data operations and manipulations that pertain to how the end user will interpret or interact with the data, but not the operations and manipulations needed to store and retrieve the data efficiently.

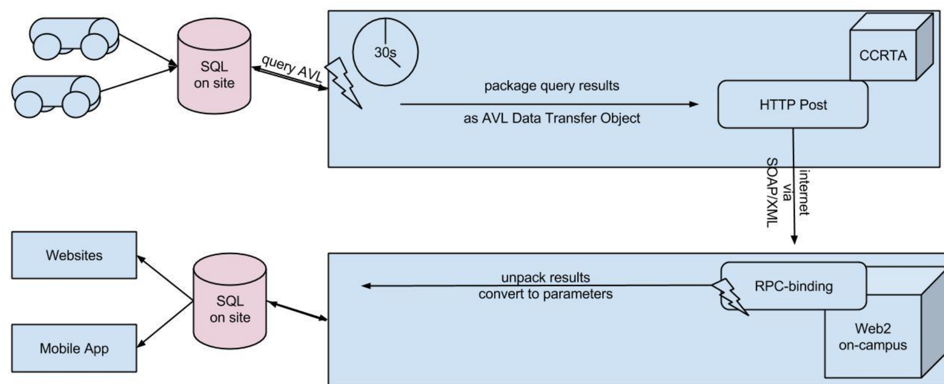

Figure 5 A graphical depiction of a simplified BusLocator

[22] The qualification of data that is difficult to analyze by traditional or non-automated methods due to its sheer volume.
[23] Simple Object Access Protocol. A specification for exchanging structured information via message based web services. Implemented by an XML structured definition. Consumed by messages with a distinct metadata header and data body.

Next Bus ETA Helper

Next Bus ETA Helper was the resulting work of the summer 2013 portion of this project. Microsoft training technology[24] was used to best understand the how to replicate and improve upon the timer-based concurrency system of the Bus Locator windows service. The purpose of this project was to provide the same data fetching functionality of Bus Locator for ETA data provided by a third party. This windows service would periodically poll Next Bus' XML web service for ETA data instead of consuming SQL data over a local network. Special considerations were taken to determine:

- What is the most appropriate time interval between polls
- How to pair ETA to corresponding AVL history for analysis
- How to reuse this mechanism for providing ETA locally to GeoLab applications

The interval chosen between pull intervals was two minutes. The smallest unit of time considered was 30 second intervals since that was the rate BusLocator pulls AVL data and sends it to NextBus for use. This reference measure was doubled to account for a 50% drop and resend rate worst-case over the network. That unit of time was doubled again for two reasons. A unique pull request had to be made for each stop in the RTA. The average process rate to obtain the entire body of data for each of CCRTAs active stops clocked at thirty seconds per batch request. Each pull was threaded and this time interval was measured from the start of the first request thread until the end of the last. Organization became difficult because of this restriction as each timer call was a thread itself. It was a concern that if a batch request exceeded two minutes, and then it would overlap with the next pull request causing a backlog and aggregation of tasks. This could become an enormous performance problem. Two minutes was measured as the minimal amount of time to not create that overlap – rounded up to the nearest 30 second interval.

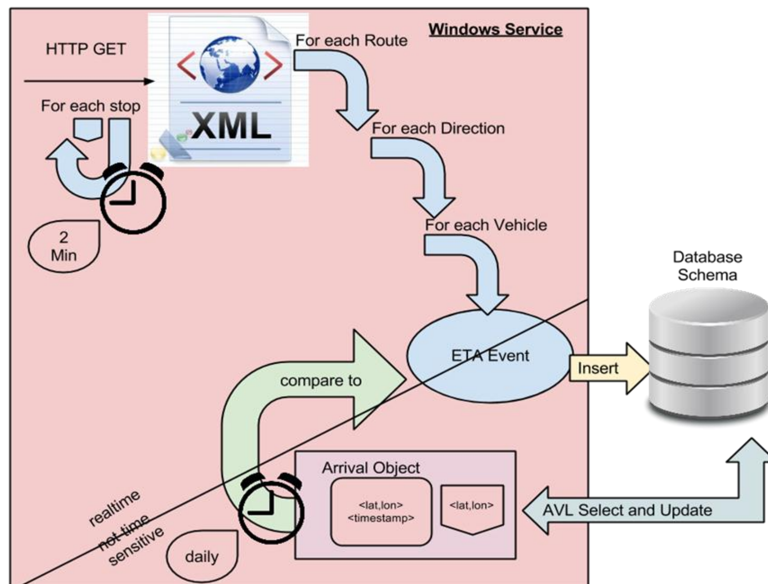

Figure 6 The data flow of ETA data collected and analyzed by NextBus ETA Helper

[24] Provided by BSU through an ATP Summer Grant.

The data lifecycle was difficult to guarantee in this model. Event-based concurrency with a reliance on network communication is more appropriate for tasks that are small and performed infrequently. Initial attempts to pull data resulted in missing points or pulling redundantly and causing network congestion. In trying to combat these problems in the creation of GLaaS, two techniques were researched:

- Exponential Moving Average
  - Used to determine pull interval in a more robust system
  - Creates a more flexible system that can handle fluctuating performance time of tasks
- Dataflow and Actor Model concurrency
  - A way to break up tasks into fault tolerant, scalable threads
  - Improves performance by scaling resources to match back log and bottleneck

The intention of using these algorithms was to best approximate the rate at which the real-time data was produced. Minimal time between production and request and minimal redundant data pulls was sought.

GTFS and NextBus

While the previous parts of this section have been focused on real-time, data structures, static data often gets overlooked for being innovated and automated. A recent grant between the GeoLab and MassDot resulted in the creation of GTFS databases for RTAs in Massachusetts. These GTFS files are CSV[25] organized text file databases that list particular aspects of transit information. These collections of data are considered feeds, because they can be consumed by Google or third party developers on demand. In practical application, however, the transfer time and data involved is updated infrequently compared to rate of use by patrons viewing the data on google maps. The purpose of these feeds is to provide a computer readable form for common RTA scheduling data such as routes, trips, stops, shapes among other metadata. The data can be automatically added to mapping applications, and users can connect routes, trips, et. al. between several RTAs to get from point A to point B efficiently.

The consumption of GTFS is automated while its creation must be custom fit by scheduling software or encoded manually. Efforts have been made throughout this project and the projects of other research students to automate the creation of GTFS. Discoveries of using third party tools, GIS file types and standards, and the creation of input applications has all been attempted and accomplished with varying degrees of success. When expanding upon GTFS for purposes of providing CCRTA's scheduling data to NextBus via a common standard, the working title NBFS was given to the feed. NextBus had made use of the GTFS model for their ETA calculations when comparing AVL data against the static schedule to determine lateness. Two optional fields in GTFS were required for them to do this calculation. The need for a collaborative controller became apparent while developing tools for static data such as GTFS and NBFS. The majority of the work was done manually. If programmed and

[25] Comma/Character Separated Values. A plain-text representation of a spreadsheet or 2D array of data consisting of rows and columns. Typically, the initial row defines the data fields for interpretation by human or computer. http://tools.ietf.org/html/rfc4180 -- An Internet Engineering Task Force Definition of CSV.

automated by a third-party service, then the knowledge of this work could be reused free of application context.

Results

Pull design pattern and improving transmission quality

Unknown variables that can inhibit data transmission are minimal when a single developer is able to make both the production mechanism and the client application or consumer. Researchers are able to measure and make public their production rates and client systems can use that information to determine the rate at which to request data. The producer and the consumer are not created by the same entity in most cases. It is uncommon for production rates and statistics to be generated and stored with the real-time data. These measures are typically for system auditing use, and do not have an on-demand application internally. In these cases, it is the responsibility of the client to produce a reliable statistic that measures a satisfactory pull rate.

When data must be requested of a producer, and is only given on demand, this is sometimes referred to as the pull design pattern. It is also known as the request-response pattern, or the query pattern. It commonly is invoked when an application requests information from a database or shared resource. The following were examples of the pull design pattern from the: BusLocator pulling AVL, ETA, and Route data from trapeze; Next Bus ETA Helper pulling xml ETA packets from NextBus; and GTFS-realtime specifying RTAs and RTA data providers host data at a public URI[26].

On-demand usage of the pull design pattern

On-demand applications that consume real-time transit data do not concern the issue of pull rate. An example is a text-based website that gives the most recent AVL data in textual form upon request. This application does not need to actively update data in real-time, because it is agreed that they are bound to the user interface. In this example, the application's data is bound to the most recent data available at the time of access.

Consider figure 7. Here is an application for the given example; this website is displaying AVL data in text based form. Note the three outlined parts. The first is the contract with the user stating that this data is the most recent available for a particular time. The second is a User Interface binding in the form of a button. Using this button, the user can make a new pull request and demand more data. The third is the graphical representation of the AVL data with geolocations as approximated street addresses[27].

[26] Uniform Resource Identifier. A text string used to identify a web resource, file or location. Interchangeable with URL or URN.

[27] This process is called Reverse Geocoding, where geolocations are compared to a street address region by a third party database provider, such as the Google Maps API.

1 Information Retrieved at: 11/30/2013 5:20:46 PM

2 Refresh

☑ CCRTA Fixed Route ☑ CCRTA Paratransit ☑ Plymouth & Brockton ☐ Peter Pan

CCRTA Fixed Route Vehicles

3

Route Name	Bus ID	Time	Speed	Location	Heading	Direction
Villager	44	5:20 PM	0 MPH	Hyannis Transportation Center (HTC), Barnstable, MA 02601	N (0°)	OutBound
Villager	58	5:20 PM	13 MPH	3195 Main Street, Barnstable, MA 02630	SE (135°)	OutBound
H2O	39	5:20 PM	35 MPH	East West Dennis Road, Dennis, Barnstable County, Massachusetts, 02639	S (180°)	OutBound
Sealine	41	5:19 PM	0 MPH	32 American Way, South Dennis, MA 02660	N (0°)	InBound
H2O	55	5:20 PM	31 MPH	371 Main Street, Harwich Port, MA 02646	W (270°)	InBound
Sealine	9	5:20 PM	0 MPH	12 Bertram Avenue, South Dennis, MA 02660	N (0°)	InBound
Flex	11	5:20 PM	31 MPH	273 Great Western Road, Eastham, MA 02642	W (270°)	InBound
H2O	10	5:20 PM	0 MPH	Stop & Shop, Orleans, Orleans, MA 02653	N (0°)	InBound
H2O	3	5:19 PM	0 MPH	12 Bertram Avenue, South Dennis, MA 02660	N (0°)	InBound
H2O	12	5:20 PM	0 MPH	75 Yarmouth Road, Hyannis, MA 02601	N (0°)	InBound
Flex	14	5:20 PM	0 MPH	100 Shank Painter Road, Provincetown, MA 02657	N (0°)	InBound
Sealine	16	5:20 PM	0 MPH	Steamship Authority Ferry Docks, Woods Hole, Falmouth, MA 02543	N (0°)	InBound
Flex	18	5:20 PM	0 MPH	16 Sisson Road, 333 Route 28 Shopping Center, Harwich Port, MA 02646	N (0°)	OutBound
Sealine	19	5:20 PM	13 MPH	Steeple Street, Mashpee, MA 02649	W (270°)	InBound
Sealine	21	5:20 PM	0 MPH	Hyannis Transportation Center (HTC), Barnstable, MA 02601	N (0°)	OutBound
Sealine	22	5:20 PM	13 MPH	694 Falmouth Road, Mashpee, MA 02649	SW (225°)	OutBound

Figure 7 Application example of on-demand, pull design pattern for transit data.

Real-time usage of the pull design pattern

Unintended side-effects occur when trying to use the pull design pattern on real-time consumption services. The most used GeoLab application that fits this description is real-time mapping.

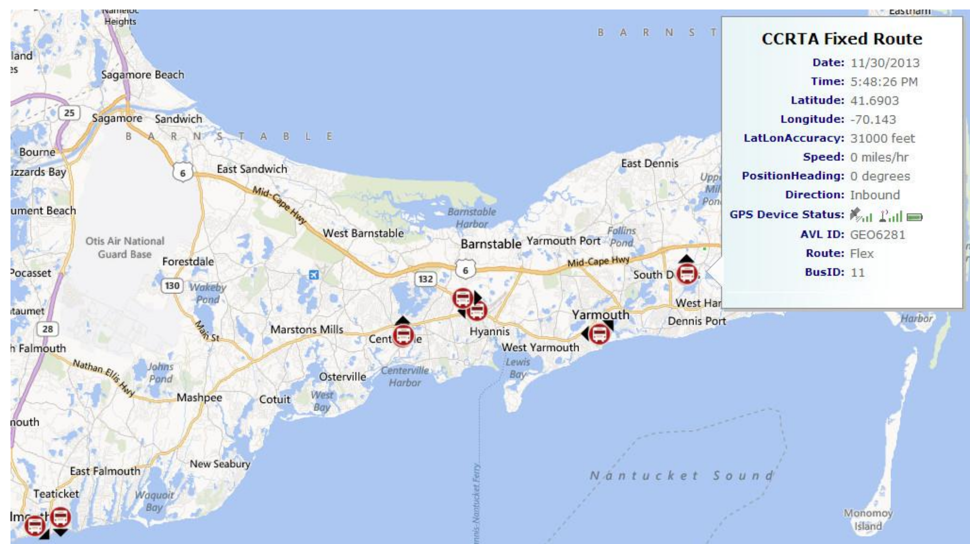

Figure 8 Real-time, mapping application that uses the pull design pattern. It displays the most recent AVL data for CCRTA vehicles at 5 second pull intervals.

There are two side-effects that have different costs. The first occurs when the client pulls too slowly. This means that real-time data points are request more infrequently than they are created. The client is not consuming the all of the available data. This is not apparent as a problem in real-time display and

use. As previously described, patrons do not notice small delays in AVL or ETA. But when compounded with the discovered possible problems of network reliability, database collision, and other issues, a dropped packet can cause a long interval of data loss. When going back to information stored by a client for analytical purposes, this issue can make data sets unreliable. The second side-effect occurs when the client pulls too frequently. The simple problem, occurring on the client side, of pulling redundant data can be solved by only storing unique data points. The larger, hidden problem is overloading the network on the server side. By making redundant requests, the client is causing the server to do work. In most pull systems, over network via http or web service, the server does not know which client is pulling it. Even with using cookies or other persistent connection techniques, the server must do work to guarantee that it only does unique work. It should also not be the duty of the server to break its response contracts – e.g. returning the set of most recent AVL data – by omitting redundant points since the server is not aware of how the data is being used or consumed. It is the duty of the client to address these issues.

Over the course of this project, alternatives to the pull design pattern were discovered. These include the push design pattern, sockets and websockets, and the subscription or observer model. They have several implementations and are part of the multimodal service layer of the GLaaS API. To find a short term solution for systems that can only implement the pull design pattern, such as bus locator, an alternative algorithm was employed to help minimize these two side-effects.

Exponential Moving Average and approximating optimal request rate

The *exponential moving average* is a function that can be used to determine trends. The pull interval can be determined dynamically based on the average interval of actual real-time data occurrences using this function. In general, the function can be written as:

$$\tau_{n+1} = \alpha t_n + (\alpha - 1)\tau_n$$

τ is the predicted time or value of the average. t an actual or recorded time or value that occurred. n+1 is the next occurrence we are trying to approximate and n is the most recently recorded occurrence. A simulation as created for this project. Its purpose is to determine if there is general improvement over constant rate pulling. Since BusLocator and pulling actual data would cause too many unknowns for reliable results, vehicles were simulated based on their observed behavior and record. A collection of 10 vehicle threads as c# timers were made. Each vehicle thread pings a shared resource object, implemented in a synchronized array of AVL points, to simulate populating a database record of the most recent data for each vehicle. Each vehicle would ping at a randomly chosen rate between 5 seconds and 15 seconds to simulate observed GPS ranger behavior over wireless signals on Cape Cod. Vehicles were also given a 10% drop rate for each packet to simulate network congestion and collision between ranger and shared resource. No noise was simulated between Client algorithm and shared resource.

Two algorithm implementations were started at the same time. One was a simplified version of BusLocator's pull algorithm labeled ConstantRateConsumtion. It would pull the most recent data available to the shared resource at 30 second intervals. The second algorithm was an implementation of

the exponential moving average. A proportion of a constant .5 was chosen for α as to be a fair estimation. Since data points were pulled for all vehicles, and not each vehicle independently, the occurred time, t, was calculated from the average lag time between the AVL points and the previous predicted interval. Specifically:

$$t_n = previous\ interval - average(vehicle.datetime - current.datetime;\ for\ each\ vehicle)$$

While this may seem logically recursive, it was a simple way to implement the Exp. Avg. calculation without memorizing the intervals of each vehicle. Vehicle AVL points are pulled in batch, so the difference between the AVL occurrence and the present was calculated. The average of those differences was subtracted from the previous interval prediction instead of comparing that datetime to a previous recording of the vehicle AVL history.

For example, 10 vehicles were pulled and the average difference between then and now was 5 seconds. If the previous pull interval was 25 seconds, and we were on average 5 seconds late, then the average interval for all the vehicles at the previous time must be 20 seconds. This decision was made because vehicles change often in practice application. It is certain that recorded instances would be improved from caching vehicle history for at least two iterations. The one issue that occurred with this calculation as opposed to caching was: if this algorith's rate of pull is too quick, then we would be calculating based on data for the (n-1)th time, not the nth. So, if the average difference exceeded the previous interval, then the previous interval was too quick. So the actual occurred time, t, is the negation of the difference – which becomes a positive value) – that is added to the interval. The pseudocode for the algorithm used can be seen here:

```
alpha = 0.5
For each (AVLPoint point in most recent AVL from shared resource)
            {
                if (point is null)
                    continue //exit loop and continue, data is not ready

                diff = current_datetime - point.datetime
                if (diff < sender.Interval) //we are pulling to slow
                    sum += diff // make the sum bigger
                else //we are pulling too fast
                    sum -= diff - sender.Interval //make the sum smaller
            }
            t_avg = sum / number of vehicles

            Interval = (alpha * (sender.Interval - t_avg)) + ((1 - alpha) * Interval)
```

In perspective, you can also consider the constant rate pull algorithm to be the same as the exponential average, with $\alpha = 0$. This would then only consider the value of the previous interval when predicting the next.

Results from running this algorithm addressed the first issue the pull design pattern has on real-time data systems. The exponential moving average algorithm better approximated the pinging intervals

of the vehicles and dropped fewer packets. Because of not pulling each vehicle individually however, as is the design of this example system, results saw an increase in busy waiting and redundant pulls. The burden was placed on the server rather than the client in this instance.

Simulation Summary	Constant Rate Alg.	Exp. Mov. Avg. Alg.
Total data pulled	3009	14819
Unique AVL Points Pulled	3006	8126
Redundant AVL Points Pulled	3	6693
Missing/Dropped AVL Points	5169	49

Figure 9 Summary of Exp. Mov. Avg. Algorithm simulation on 10 vehicles.

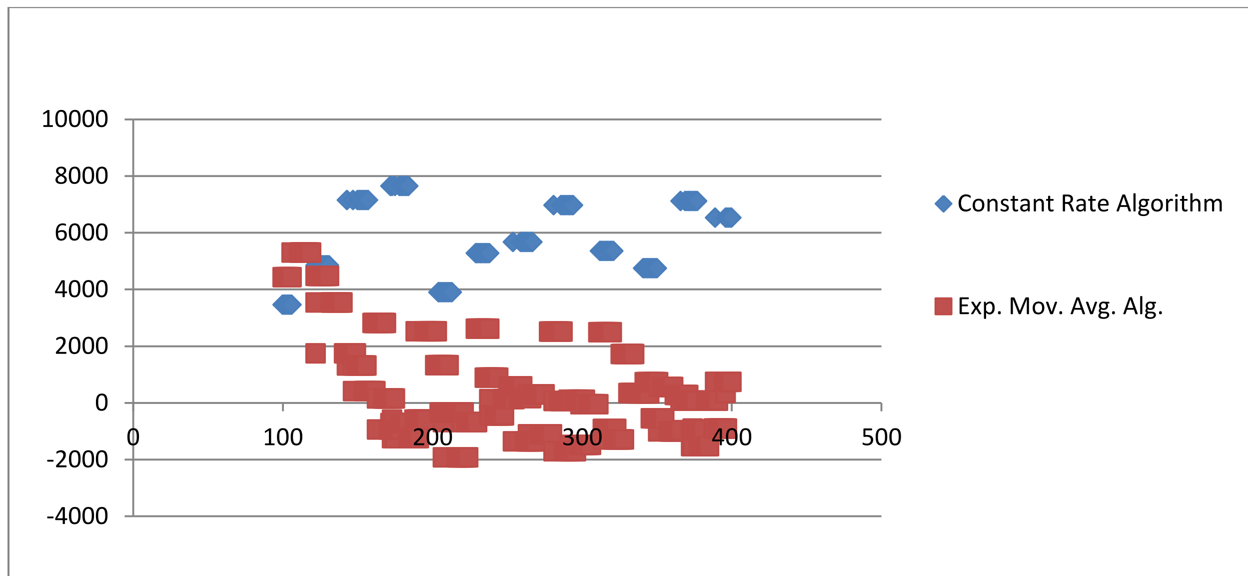

Figure 10 A point chart of pulled AVL points from both algorithms between AVL id's 100 and 400

Consider the chart in figure 10 as a graphical representation of a 300 AVL point sample. The horizontal axis represents unique AVL points. Each AVL point was given an id that represents the unique pairing of vehicle id and datetime. The vertical axis represents the difference between the average t_avg and the predicted interval τ measured in milliseconds. A negative value on the vertical axis means that the predicted value was too fast when compared to the occurrence. The density of points, measurable by multiple points sharing the same x value, is the result of redundant pulls. Notice that the exponential moving average's results are densely populated. The constant rate algorithm is not. An algorithm that best approximates the ping rate of real-time data would stay close to 0 on the vertical axis. The optimal case would be points that come from the function:

$$y = 0$$

In the optimal case there would be minimal difference, approximately 0, between occurrence and pull, and there would be no redundant pulls requests. Future implementers of this algorithm should adjust the value of α dynamically based on success. In practical application, vehicle rate variation would be smoother since those variations come from geographic occurrences such as distances and weather, and not pseudo-random chance generated artificially. This algorithm is a smoothing algorithm and would better suit gradual change that is more consistent.

The optimal solution does exist, but its implementation is not in technologies that can only perform the pull design pattern when requesting data over network. A connection-oriented design pattern is better used for this purpose called the push design pattern.

Discussion

Security and the closed pipe vs the open pipe

When applying the MVC model to these pull based applications, typically the access to the data is secured at the server side. Since access to data is done explicitly by the applications themselves, there is no need to have a policy for which systems and users can call specific procedures. When moving to an open request model that this project has dictated, there is a need for such a protocol. The security responsibility is no longer on the storage mechanism itself, but now on the serviced functionality that manipulates the data. The transit data that this project utilized for its findings is not sensitive to any customer or private information. For good practice it is preferable that the systems being built for open reuse have the consideration of security. Examples of two methods that are easily used by these solutions for data migration are *authentication* for private access and *differential privacy* for open access.

Authentication

Authentication is a basic technique of web security. In the context of transit data, it is important to guarantee that data integrity is not compromised between storage and retrieval. In the current systems, protocols specify a non-committed approach to the data retrieval. TCIP states that the XML production and consumption of transit services is done through approved vendors, and not designed for open access. GTFS and GTFS-realtime has information posted to a public resource URI, which requires anonymous authentication by default. To be independent of these constrains, the protocol for security used in this project needs to be flexible.

In the data chain, it is important that the data structure stay free from the implemented context. If access was restricted at this level, then each application would need independent credentials for each data type. Access would be general to an all or nothing approach, where applications either have read access or they do not, or they have write access or they do not, etc. The does not allow for fine tune control of access specification. Abstracting authentication to the access level, either stored procedure or application level access such as an API[28], can ease this problem. Now applications can have a finer granularity of access of data, but specifying what approved operations they can perform. This

[28] Application Programming Interface. Either a library or a service that specifies how software components or functionality should interact with each other and be used.

authentication should be done before the call is made, to reduce unnecessary work. By abstraction authentication to this level, it is now guaranteed that application access does not invalidate the integrity of our data structure. These considerations guided the protocol decisions for the GLaaS Model and API.

Differential Privacy

There should be another methodology for insuring security in an open system. The purpose of defining these methods was to insure the fastest and easiest way to connect application to data migration techniques. In moving to this open approach, it is a goal to still produce data that is consumable in multiple forms, independent of use. The problem that comes along with these improvements is similar to problem with big data repositories and survey or customer information data.

As the data becomes more accurate and granular over geographic regions, periods of time, and associated services, consumers of the data service are able to notice patterns in data feeds. These patterns are an important part of the analytical repurposing of real-time data. For example, analyzing vehicle AVL history for non-stop points can be a leading clue to where the next bus stop should be placed on a given route. This can also have unintended side effects. Christine Task describes that in collecting large amounts of survey data, customer information may still be obtainable even with basic anonymity measures in place.[29] This pertains to surveys where name, address, and other identifying information are removed and only the survey-relevant statistics are scored. With a sufficiently small sample size, it is possible to use marginal distribution to find out who a person is by making correlations between their answers and outside information. The problem arose from a participant answering both about student demographic information – such as major, minor, etc. – and about experience with drugs. This is similar to a problem in public transit AVL tracking. As we use the above methods to improve accuracy and relay of AVL information on paratransit and non-route vehicles, we may be uncovering patterns unwillingly about individual ridership behavior. Where a vehicle is at any given time is acceptable to be displayed and updated in real-time in a map application or a text based representation without persistence. For analytical queries, which should be possible by repurposing the data model, this may become an issue for a public facing data feed. Cynthia Dwork, scientist at Microsoft Research, provides differential privacy[30] as a solution. The concept is to provide just enough accuracy of data, determined at time of access, to provide purpose to the request but minimize the difference one data point makes on the overall information. When repurposed for AVL and geolocation based data, this method is similar to selecting a geofence – or a geographical zone – around the AVL point. The point is accurate inside this geofence to a specific margin of error. Until the vehicle moves outside the geofence, no new data points are provided for analytical consumption – such as in a pull request of AVL history for a time period. In order to protect customers who use transit services, a geofence should be chosen inversely proportional to speed for non-route vehicles. As a vehicle slows down, the uncertainty fence expands. Requests for a slow vehicle (or stopped vehicle) have a large range of uncertainty between concurrent days. This obfuscates the particular house that a paratransit vehicle stops, protecting the rider.

---

[29] Task, Christine. "An Illustrated Primer in Differential Privacy". *Crossroads* Vol 20.1. Fall 2013.

[30] Dwork, Cynthia. *Differential Privacy*. http://research.microsoft.com/pubs/64346/dwork.pdf

Conclusion

Design considerations when moving forward and making GLaaS

These observed patterns and methods for transporting transit data came from analyzing the current system. It seemed that passive documentation may not be enough to provide insight into how they work. GLaaS was designed as the last part of the project. It serves as an expandable and reusable system in which these ideas could be designed and tested, without breaking existing functionality. From the core concept of keeping data and application logic separate, two main parts were created: GLaaS Model and GLaaS API. The model would be the reusable structure, definition, and protocol for all transit data structures used by GLaaS. The API would be the implementation of the above algorithms and the manager of all user level authentication and privacy enforcement. The Model would only be accessible from the API in order to best insure data integrity. The API would have multiple modes of access to best support the different data production and consumption rates. It would also support multiple rates of consumption of the same data, simultaneously. Lastly, purposeful patterns and documentation would serve as training for maintaining, replicating, and extending features of this project.

GLaaS Model and Verification of Database Design

Research Questions

Is there a way to use the data structure definitions to create a data management system that is both reusable and easy to maintain? What is the easiest way to do that so effort, documentation, and structure is sufficiently clear, concise, and reliable?

Purpose

It was apparent that the techniques studied so far in this project could be directly applied when maintenance BusLocator after a transition the WCF portion to a new webserver. Documentation and data structures from each project that relied on the AVL data were separate and only accessible from their parent applications. Data stored in persistent forms was not easily movable to other applications without being parsed and versioned. Points of interception where efficiency and analysis algorithms could be implemented were not apparent and varied between systems.

As previously discussed, the current Data migration programs used in the GeoLab are aptly suited for their designed purpose. The intent of designing a new, reusable database model for transit data structures is to gain the benefit that a few levels of abstraction provide. The first is that by separating how data is stored and retrieved from how it is used and manipulated, those two pieces and run concurrently and with separate algorithms and efficiency measures. Consumers and producers of the data do not need to know where and how the data is stored, but only how to interface with it in an approved way. The duplication, preparation, and caching of data can be purposeful to offload the burden from the client. Lastly, features and modifications can be separated and issues will not cross over between existing systems.

Consider the following example. Researcher J wants to make a new mapping application that incorporates real-time AVL data from a known source with that of a new one to make a robust, multimodal map. In order to do so, it may seem easy if a current map already exists. However, currently the GeoLab has databases designed for their intended use, and not for interfacing with their possible data. So, each vehicle has its own data table and is easily retrieved from the current map. For J to achieve their goal, they need to either duplicate the structure in its entirety to best avoid conflicts with the current system or they need to find points of inject at the data level that they can insert their code and not affect the existing performance. Both of these approaches have problems that stem from the underlying structure.

The solution is to define a database model and feature off of the manipulation of the transit data structures defined in this project, and not derived from their intended use. An application level API or service can interpret the data in multiple ways for general and specific use. By making this separation, the integrity of the data can be kept while different applications can interface with it in parallel. The pattern followed for this particular implementation is MVC – Model View Controller – but the View is left unimplemented except for in tests. A client application was not the goal of this project.

Methodology

The database schema is designed in two parts. The first part is a strict protocol for Table creation based on *features*. A feature can be thought of as a transit data structure or as any other data structure from which applications can be based on. The two example data structures that have been created for this project as examples of the protocol are AVL and Event Logging (abbreviated Log). The second part is the approved creation and modification of functionality at the database level. This is implemented as Stored Procedures, which are SQL level functions that can take parameters and return tables, vectors, and scalars or simply perform work.

Initial Design – triggers and the trickle down structure

Table Structure

Each Feature would have three types of tables: a primary table, a set of specific tables, and a set of info tables. Each table has a particular purpose for aiding in keeping the data lifecycle purposeful and keeps unnecessary read and write operations to a minimum.

Primary Table

The primary table would be the contract resource for the entire feature. Transit data would be inserted into the primary table upon first entering the server via insert calls. The table would be unindexed to allow for quick insert, but would have a unique, auto-incrementing primary key. This means that every transit data object in the primary table would be unique upon insert. If data were to be reinserted, it would have to be known by its primary key, or deleted and reinserted. If the data was significantly changed then there would be no need to delete and the reinserted data would be sufficiently unique.

The primary table would have required fields for each of the minimal definitions of the transit data structure. For example, an object must have a vehicle id, a geolocation, and a datetime in order to be a valid AVL point. Optional fields for each table would be the commonly used or purposeful metadata that could accompany the transit data structure on insert. For AVL this is commonly speed, direction, route information, block information, and other static, dynamic, or composite transit data structures. This is distinct from the previous organization structure, as AVL tables may have had required fields that were simply metadata. Making metadata required prevents reuse for systems and purposes that do not contain the same metadata, requiring application developers to create new parallel structures. Application developers with metadata that does not exist in the current feature's primary table can simply request student researchers to add support for that data. The student researcher can do so by adding an optional field to the primary table, which is guaranteed not to break any of the existing functionality if the protocol is followed. We have now minimized time of adding and modifying existing structure, while not having to maintenance existing systems as a result of the change.

Specific Table

The specific table is the closest each feature should get to business logic. It is the structure and ordering of data from the primary table that best fits a particular access mechanism or operation. It is still separate from how the data will be used, but provides a convenient structure in an attempt to

provide efficiency and clarity for researchers wishing to build upon the existing functionality. For example, a common operation that is performed for AVL data is selecting the most recent AVL data points for each vehicle. This could be pulled easily from the entire history of AVL, selecting a cutoff of a reasonable time, and indexing the table by datetime and then by vehicles. This query is expensive for large datasets, even with proper indexing. A good specific table for the feature that could alleviate this problem would be a table that contains one row for each unique vehicle id that has been entered and only stores the most recent AVL data point for that vehicle. Since vehicle ids would be unique they could be considered the primary key. Selecting the most recent AVL for all vehicles is then a simple select of the entire table and selecting the most recent AVL for a particular vehicle is a selection against a clustered, unique index. Both of those operations are fast and cheap. There is a cost associated with updating such a table, but it is no more expensive than inserting into a large non-unique, indexed table on several columns. The second benefit is that the purpose of this table is clear, and can be reflected in its name. Multiple applications can use such data, and expanding it to support new metadata and applications is the same as the primary table. The database designer now gets to make decisions from the question: “What is best for the database? “rather than: “What is easiest for the current application?”.

Lastly, specific tables must have all the required fields of the primary table, but since they are data delivery specific they do not have to have all of the optional fields. If two specific tables were made to make a distinction between vehicle specific use and rail / train specific use, we would have a set of train specific AVL fields and vehicle specific AVL fields. The rail specific table only needs to implement the optional AVL fields corresponding to being a valid train; the vehicle specific table only needs to implement the optional AVL fields corresponding to being a valid vehicle. This distinction does not violate the protocol of organization instead of use. It is not being dictated what kinds of application are going to consume this data. It is instead that the AVL objects are being extended to be more and more specific, and at some point are better collected in one way than another. Now there would be tables that can be organized, indexed, and modified based on mode specific parameters, but can be consumed by a variety of context-free applications.

Info Table

The info table is a lightweight, compromise between the above table structures. It is an indexed, single purpose table that contains metadata and relationships for transit data structures by id. It can also be used to be the primary insert location and select from location without the delegating structure above, given that the data is both inserted and selected infrequently. The Info table structure is a way to also update transit data points after the fact, when editing the specific table is either unnecessary or messy. For example, consider an info table Info_AVL_Late. This table simply contains a collection of AVL ids of AVL points of vehicles that were late. That calculation is done at the application level, and we did not want to burden the primary table and each specific table with this very simple data. The use of join operations in SQL can assist us instead. It is assumed that the AVL points in question are already gotten from one of the specific tables. Then, that data set can simply be joined with the Info_AVL_Late table where their IDs equal to find out which data points are late.

It might be difficult for researchers to decide whether to add an optional field or to add a joinable info table. The protocol made from this project does not penalize the choice, but recommends that if the metadata is not readily available upon data point creation, then it should be an info table. Lateness cannot always be determined at the same time the AVL point is made, and might be calculated later. That is why it is better suited as a separate info table. Information like block number and route can likely be determined at creation time from the source, and can be optional fields of the primary and specific tables.

Stored Procedure Structure

Prohibition on application level queries

It is easy to think that at this point the protocol should relinquish control to the application level. There is one more level of indirection required by the GLaaS protocol, though. All interactions from outside the database must be performed by stored procedures. The purposes of this rule are to minimize maintenance and debugging and to avoid security risks of open systems. In the current GeoLab systems, application specific stored procedures have been created and as a result some client applications have made *work-arounds* or temporary procedures to fix lacking functionality. These procedures are designed at the client application level and execute dynamic SQL queries that are created as plain text. There is nothing inherently wrong with this design, but in maintenance it has become difficult to debug the application and the database at the same time if it cannot be assumed that the problems are localized to one or the other. Application problems may masquerade as database problems by using these rogue queries. By imposing that application level requests must be made through stored procedures the problems that are being solved are: fixing a stored procedure once fixes issues for all dependent applications, database issues are traceable and application issues do not supersede their boundaries, and the parameters and operations that can be performed on a data structure are explicitly clear.

The hidden benefit is also one layer of indirection between security attacks and sensitive data. Consider for example ridership information and a client application that inserts and selects client information that is open to the public. Assume that permissions allow anyone to insert, but only authenticated users could select data at the application level. An anonymous user might put some rogue data in the database, and we can assert that it meets the requirements of creating a valid account before inserting it. If we allow the application to make a custom, dynamic query at the application level then they would likely be incorporating the data into the query definition:

```
public SQLTableType openCreateUserMethodWithoutValidation(string username)
{
        return Sqldelegate.executeQuery(
        “Insert into Primary_Users(username) values (‘ “ + username + “’);”
        + “Select * from Primary_Users where username = “ + username
        );
}
```

Figure 11 An example of an application level query in a c/java/c# like syntax that can lead to SQL injection.

The rogue user could enter in a string such as:

```
someUsername'); Select * from SELECT * FROM information_schema.tables --
```

This is a typical technique of SQL injection where a user injects SQL at the application level and it is bound to the query at compile time. This string in particular would enter in the someUsername value, end the query, select information about all the tables in the database, and then comment out the rest of the query. If the application level does not have proper checks, then the integrity of the database can be compromised. Instead of returning relevant data about the user to the user, the tables that belong to the database are returned. If successful, the rogue user could repeat the process and perform serious and possibly irreversible damage to the database.

This is prevented when only allowing stored procedures. Data content is bound at runtime in the SQL environment when passed as parameters. If the same technique was used with a stored procedure, the data would be packaged as a string literal and get inserted into the field as a whole, not interpreted as a static command. This would also leave a trail for database administrators, as rogue users would be the source addressed of queries that contain this malicious data. Since the data cannot actually harm the database in this way, it would simply either be inserted into the table as a value, or logged.

Stored procedures

Stored procedures in this model function similarly to in the current system. There are only a few added rules of their creation. There should be one point of entry into the primary table. It should be structured as an insert query that takes all parameter. This includes both required and optional parameters. It then sets the optional ones to null in order to let executors of the insert not set the optional parameters. It provides and data validation which includes but is not limited to: asserting required values are not null and of the correct data types and asserting values entered do not violate any uniqueness rules of the data structure. The data is then inserted into the primary table. In this initial design, triggers were used to

Triggers

Triggers are automatic queries that attach to database objects such as tables and execute upon the occurrence of particular events. The GLaaS Model was going to use triggers as an efficient mechanism to establish a *trickle-down* structure between primary tables and specific tables. A trigger would be made for each specific table that takes data for that feature. The triggers purpose would be to:

- Evaluate if the inserted data is appropriate for the specific table
  - The data must have the correct optional fields for the specific purpose
  - The data must be unique by the specific tables uniqueness rules
- Insert the data into the specific table
  - Or update existing data objects in the specific table with values from the new insert

It was logically assumed that data insert and update for each trigger would be concurrent since each query was contained in its own trigger definition, and specific tables were not codependent. Each specific table only needed to get its data from the primary table or through joins with info tables.

Issues with triggers arose when the inner working of trigger logic was revealed. Triggers are not concurrent, and how they are executed is contrary to how they are created and stored. When an event is raised on a database object, such as a table, that has a trigger listening for that event:

1. The original query is suspended until all triggers are complete
2. The trigger query is statically attached to the original query
3. Each trigger is performed in order, and in single threading
    a. One trigger's query cannot begin until the previous one finishes
4. If one trigger fails, then the whole query fails, and the original insert and all trigger queries are undone.

This realization greatly conflicts with the principles of this model and protocol. This means that specific applications of a feature cannot only interfere with each other, but also invalidate the integrity of the original data structure. A new methodology was sought to achieve a similar affect, without the dependency issues.

Subsequent Design – Batch Query

Solutions were tested to try and resolve the trigger dependency issue. It made sense to simplify the structure and rely on existing objects provided by the protocol rather than having a difficult or complex structure from the beginning. The functionality the triggers would have provided were moved to the insert stored procedure. Batch querying through query delimiting was utilized to let the SQL environment best decide concurrent actions for itself. The output functionality of insert queries was used to replace the trigger siphoning function of inserted data on the recommendation of several online SQL training resources and Microsoft's Plural-Sight developer training. The first insert is still performed on the primary table. The output filtered values are then inserted into each subsequent table. Each specific query is separated by the delimiting character. The semi colon ';' is the delimiting character in this implementation. This instructs the SQL environment to execute each query in its own batch. Since the specific queries are only dependent on the output data's integrity and not on the integrity of others, the environment will allow the queries to be scheduled, compiled and executed concurrently. If the data is malformed the subsequent queries will not be executed because they all depend on the output data. Malformed data will not be able to be inserted into the table initially, and so there will be no output data to trickle down. If one specific query in the chain fails then the others will still execute, because of the lack of dependency. The downside to this design is that its function is not apparent in its appearance. It seems that these queries being executed literally in the same procedure would be more dependent and sequential than triggers all listening on one object, but it is the contrary. This design proves to be efficient at tying functionality to the data, as now if any data is malformed or entry is incorrect, the areas of potential problems are limited.

Eliminating the primary table as it is no longer listened to directly by triggers was considered. It seems better to use it as a validation tool and a contract for to enforce the integrity of the data structure. As features get increasingly specific in subsequent tables, the primary table can be the unifying structure of what possibly constitutes the data. Between an object being a functional

description of the protocol and using the table as data validation, it persisted in the final design of the framework.

Results

The tangible product of this design and protocol is the GLaaS Model database object implemented at the GeoLab. This data model for MVC applications, especially for real-time data structures, can be achieved by following the methodology. The database structure has already been defined in the methodology. The tangible result is the ability to easily explain each of the parts and to define the protocol in a way that future researchers and adopters can easily interpret. The protocol is as follows:

GLaaS Model Protocol

- Tables added to the GLaaS Model must be implemented around a particular feature
  - A feature can be any data structure such as transit data structures
- Three types of tables can be made: Primary, Specific and Info
  - The naming convention for the table is type_feature or type_feature_purpose
    - E.g. Primary_AVL
    - E.g. Specific_AVL_MostRecent
  - Required fields in primary tables are also required in specific tables
    - They are the minimal definition of what constitutes an object
  - Specific fields should be optional fields of the primary table
    - Enabling the primary table to be the most complete record of data objects for analysis and history
  - Info fields do not need to contain the required fields of a primary table
    - But they must be joinable to unique data objects
  - Specific and Info tables must be indexed on their unique or commonly selected and updated fields
    - E.g. Specific_AVL_MostRecent -> indexed by vehicle id
- Stored procedures are the only ways to interface with live data from the application layer
  - They have a similar naming convention of tabletype_feature_purpose
    - E.g. Primary_AVL_SendDefault
  - Insert stored procedures should insert into the primary table for data verification
    - Then insert into specific tables from the output values
  - Select and Update stored procedures can access specific tables directly
  - Info tables can be accessed and mutated independently

Discussion

Criteria for success

The GLaaS Model meets the criteria for good data management design and is purposefully crafted to be easy to maintain, modify, and extend. Each of these criteria can be assessed for their worth and tied to a particular aspect of the protocol.

Modification

The GLaaS Model is modifiable by design since application level logic has been separated from the data structures. Data can only be accessed and mutated in clear, explicit methods that reside in the database schema and design. Failure on one specific aspects feature is not likely to cause the failure of other specific aspects of a feature. Features are kept separate in different primary-specific clusters, minimizing data verification and missing values errors. Strict rules on what constitutes a data object can be enforced with required and optional fields. Required fields must now be chosen purposefully in order to define the data structure, and not what is required for the particular use of that data. Therefore, modification to any existing structure or function is made easier since they do not interoperate loosely and are contained in a minimal amount of specific locations.

Extendibility

The GLaaS Model is extendible by design due to the explicit partitioning of features. To add support for new data definitions of existing data structures is done by adding optional fields to the relevant database objects. Adding support for new functionality or operations on existing data is done completely in stored procedures. The guarantee that data sources must use stored procedures means adding new functionality does not require detecting data sources at the application level. Adding functionality that is robust on existing data is easy since data is associated by structure and not by use. Knowledge of tables for previous use and what they might contain is no longer necessary for developing new applications that utilize this database and its structure.

Replication

Replicating the existing model for new features and data structures is simple since a precise, but flexible protocol has been given. By agreeing to follow the protocol, developers are gaining the knowledge of this section is consistent for each database object in the GLaaS Model. Data can be shared freely between GeoLab applications without knowledge of their original purpose, and reuse of previous data knowledge is not lost.

Conclusion

The GLaaS Model is not in production use yet, but its operations are being trained and documented for other researchers continuing in the GeoLab to make use of with their projects. The benefit will be when researchers at the GeoLab adopt this protocol and model; they only need to add the logic for their project once to collaborate with everyone. Until now, application specific database logic has been created to assist in persistent storage and assessment of transit data. Over the course of this project, many different data parsing applications have been seen. GTFS, Wifi+Geolocation, train and

rail data and other data structures have been used for application purposes. Since the researchers have been working independently, these applications are robust and purposeful, but the knowledge of each application has to be relearned if a subsequent researcher wants to make use of this data. This is the same problem that occurred at the initial steps of this project when analyzing AVL and ETA data. It is urged that while GeoLab researchers continue their individual projects, they make their persistent data stores as part of this framework by adding a feature for their data structures to the GLaaS Model. Their personal application can make use of the data, but also other researchers can clearly understand, interpret, and utilize that data in their projects. Web based tools and access applications that are also reusable can be built upon that data for even more complete applications. Safety and security, data integrity, and efficiency can also be guaranteed because each researcher followed the simple protocol provided.

GLaaS API and Validation of Client-Service Architecture

Research Questions

How can the requirements of the business logic and specific applications GeoLab researchers create be satisfied while keeping the integrity and reusability of the data source? How can vendors and other specification creators be accommodated when their data requirements are tied to specific communication mediums? Is there a compromise between performance, reuse, and data validation?

Purpose

Creating and maintaining a separate, context-free data model is not useful unless an approved medium is available for application developers to interact with it. The GLaaS API is a layer of abstraction that lets users, researchers, and application developers have access to the data. Data integrity and security is guaranteed by implementing an API since each feature and operation has a point of authentication that can be created by the API designer. It was easy to perform short term operations that were inefficient and difficult to share efficient operations with other services when applications had direct access to the data model such as with GeoLabVirtualMaps and BusLocator. Through creating and maintaining a web facing API, different user levels – anonymous, researcher, third-party collaborator, administrator etc. – can be given specific levels of security that is dependent on the particular application or purpose without locking the data storage. The API is the only entity allowed to automatically interface with the GLaaS Model for this reason.

It may seem that this layer adds an extra step for GeoLab researchers to go through in order to access the data. For typical applications they would have simply written the read and write logic once in their application. Researchers adopting the GLaaS API still only have to write their logic once for the use of their application. The functionality they design must conform to fit the GLaaS protocols discussed so far as an extra cost, but there are multiple benefits. Data processed through the GLaaS API is easily stored in a persistent format in the GLaaS Model. Operations and functionality written for the GLaaS API is reusable to other applications and easily collaborative between researchers. Functionality can be written and tested separate of any particular view or implementation which saves time when deciding between several algorithms or methodologies for a task. Lastly, it is trivial to transmute data from one medium to another on demand since the client application, views, and consumers are separate from the retrieval functionality.

Methodology

Several specific considerations were made when deciding on the technology to use for the GLaaS API. The issues that have been occurring with the current systems needed to be preempted. While no technology is impervious to bugs and maintenance, functionality needed to be simple, clear, and easy to use. The intention was that mistakes, work around, and anti-patterns would be avoided if the technology made the medium simple and self-explanatory to use.

A list of criteria was made so that tools could be compared and contrasted on how applicable they were at solving these issues. Here is an abbreviation of that list in a prioritized order (from high priority to low priority).

1. The technology must be compatible with existing systems with a strong preference on the .NET Framework employed by the GeoLab. Minimal configuration must be needed to begin using the technology with existing transit projects.
2. The technology must be well-documented and supported in the .NET community.
3. The technology must be light-weight and be easily deployable on windows servers.
4. The technology must be scalable and provide design patterns for its use and adoption.

Tool choices

The two technologies that were originally compared were Microsoft's *WCF* technology and the open source project *ServiceStack.* These technologies both provide application level Webservice functionality that is an improvement on .NET web applications and web services. They are executed on the server side and both adopt the Request-Response protocol using the pull design pattern. The GLaaS API was intended to be an umbrella for most data migration applications at the GeoLab, which eliminated the possibility for using ASP .NET's MVC or WebAPI models. Those models integrate better with single, application structures and their intended use has to be manipulated to get reusable functionality. All of these technologies work well with service-oriented architecture and were possible to be used. The purpose dictates that the best technology currently available be used and scrutinized. Therefore, in order to make decision and development easier, only WCF and ServiceStack were compared.

WCF was already familiar at the GeoLab as many other applications used it for single use. It is a protocol built on RPC-message based requests over SOAP via XML. Each command a client wishes to perform is packaged as an XML document and sent to the WCF endpoint via HTTP Post. The Endpoint routes to the WCF service which unpacks the request and finds the corresponding service operation listed in the XMLs header data. The body data is turned into a DTO – Data Transfer Object –an encapsulation of serializable data types and values with no inherit functionality or methods. The service operation has been statically bound to a method by the WCF service and the DTO gets bound to parameter or set of parameters for that method. The functionality is executed and the results – if any – are packaged up in a DTO. The DTO is wrapped in an XML responses body data and the client's waiting HTTP handler is the destination of the header. The client then unpacks the request, the DTO, and can use the data inside for their purpose.

As an example, consider a WCF service that takes a RTA as a serializable, primitive text such as a string and responds with a collection of AVL DTOs that encapsulates the AVL data structure. The XML schema document (XSD) or contract the WCF publicizes to tell clients how to interact with it could look like the following:

```
<xs:schema xmlns:tns="http://schemas.datacontract.org/2004/07/AVL.Requests
" xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualifie
d"targetNamespace="http://schemas.datacontract.org/2004/07/AVL.Requests">

  <xs:complexType name="MostRecentAVLForRTARequest">
    <xs:sequence>
      <xs:element minOccurs="0" name="RTA_Short_Name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="MostRecentAVLForRTARequest"
type="tns:MostRecentAVLForRTARequest"/>
  <xs:complexType name="AVLResponse">
    <xs:sequence>
      <xs:element minOccurs="0" name="DatetimeStamp" type="xs:string"/>
      <xs:element minOccurs="0" name="Latitude" type="xs:float"/>
      <xs:element minOccurs="0" name="Longitude" type="xs:float"/>
      <xs:element minOccurs="0" name="RTA_Short_Name"  type="xs:string"/>
      <xs:element minOccurs="0" name="Vehicle_ID" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 12 An example of a WCF data contract (XSD) that is publicized to the client.

The description of the service operation (method) is set as the complex type. The parameters of the operation are inside the first sequence as elements. The response DTO is the second type contains the data structure primitives inside. An alternative documentation is the WSDL which is cross compatible with not only WCF consumers (as .NET or c# clients) but generic SOAP clients in other programming languages as well.

This technology is easy to use, but is quite hard to scale and configure. WCF requires many endpoints to be opened and hides its customizability behind XML configuration documents. Veteran .NET developers can easily configure these technologies, but new researchers in the GeoLab will have to study the technology before they can even begin to work on their application. It is easy when starting with WCF to be mistaken that configuration errors are functionality errors. *Quick fixes* and *work-arounds* might be sought in an attempt to keep up with iterative development in the grant lifecycle of projects in the lab, and initial project time can be wasted fighting the tool. After those initial steps, WCF becomes a powerful resource, but this consideration is easy to overlook. This problem is a common occurrence due to the high turnover rate of student researchers at an academic lab and can perpetuate the same issues that this project faced in its infancy. Since the GLaaS API favors using the tools and writing functional code over learning the tools and programming within a paradigm, the open source alternative was discovered and researched.

ServiceStack is an opensource web service framework built on REST[31]. The URI endpoint acts as a route directly instead of packaging the operations' route information inside the messages of the request

[31] Representational State Transfer. A service interaction architectural style focused on constraints and components over methods and binding instructions.

or the response. This means that the API is not contained as a listener on one URI, but rather a structure of URIs. The second change is that a URI can be multiplexed by its HTTP verbs. In WCF only the HTTP verb POST could be used to call functionality. GET, POST, PUT, HEAD, etc. can all be used for different operation bindings in ServiceStack. ServiceStack shuns the RPC-message model as an Anti-pattern that actively fights the HTTP on which it is built. Without delving into the protocol politics, improvement can already be seen for the two previous issues. Since ServiceSrtack can dynamically bind to routes, it can easily be replicated and scaled with minimal over-arching configuration. This lowers the perceived barrier of entry for new Lab members wishing to develop at the application level. In ServiceStack in particular, route bindings are determined relative to the host of the API dynamically above the Request DTO or at the service configuration. This provides flexibility on the part of the designer while still being explicit with collaborators. For users that do not wish to read code in order to add new functionality or modify existing operations, ServiceStack generates human readable metadata that functions as a contract – similar to WCF's XSD. The primary different is that ServiceStack does this in both a human readable HTML, and file-type specific formats. All of these formats are auto-generated and no configuration needs to be performed on the part of the researcher – save for a premade *webconfig* xml file that all ASP web applications require. What this means is that data can be consumed in methods other than XML. JSON (**J**ava**S**cript **O**bject **N**otation) is the REST favorite by accepted use and its syntax is familiar to those who use Java/C++/C#-like languages.

Figure 13 The URI route binding and auto generated metadata documentation of a sample ServiceStack project. This API services the Feature for AVL.

Consider the URI in figure 13. The first section (1) of the URI route is the host binding. IIS – a webserver management system for windows server – automates the host binding and multiplexes the web port for multiple URIs. Development and replication is maintainable since ServiceStack APIs are published as ASP .NET web applications and can be stored in the file system under this host's virtual directory. It was simple to create a virtual directory as a folder location under this host corresponding to data feature from the GLaaS Model. This made a consistent, implied purpose and connection between the API and the model. The second section (2) is this folder location that is located in the host. That folder location is known to the ServiceStack as the root address, represented "/", and all URI routes generated by the application are relative to that. This was much simpler to see than the WCF end point configuration for multiple WCF projects or making a service contract definition for each data feature. The third section (3) is the auto generated documentation url that functions as the default document for browser navigation to this page. This provides the human readable consumption guidelines for the service, as opposed to the hybrid machine-human readable XSD. Below are figures of the human readable metadata as well as the file type specific data contracts generated by service stack for an AVL feature.

GeoLab AVL services

The following operations are supported. For a formal definition, please review the Service XSD.

Operations:

MostRecentAVLForRTARequest	XML	JSON	JSV	CSV	SOAP 1.1	SOAP 1.2
SendDefaultVehicleAVLRequest	XML	JSON	JSV	CSV	SOAP 1.1	SOAP 1.2
SendDefaultVehiclesAVLRequest	XML	JSON	JSV	CSV	SOAP 1.1	SOAP 1.2

Clients Overview

XSDS:

- Service Types
- Wcf Data Types
- Wcf Collection Types

WSDLS:

- soap11
- soap12

Figure 14 The human readable metadata for a ServiceStack API with compatibility for SOAP and WCF client consumers.

MostRecentAVLForRTARequest

The following routes are available for this service:

All Verbs	/MostRecent/{RTA_Short_Name}
All Verbs	/MostRecent

Figure 15 Overridden Route definitions for this operation. Defined by the request DTO and not by remote procedure calls gives flexibility over verb usage, URI chaining, and embedding the data in the URI bypassing the need for a response object in some cases.

HTTP + JSON

The following are sample HTTP requests and responses. The placeholders shown need to be replaced with actual values.

```
POST /json/reply/MostRecentAVLForRTARequest HTTP/1.1
Host: glaas.geolabvirtualmaps.com
Content-Type: application/json
Content-Length: length

{"RTA_Short_Name":"String"}
```

Figure 16 Request DTO definition as a lightweight, serializable JSON file.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: length

[{"Vehicle_ID":0,"RTA_Short_Name":"String","DatetimeStamp":"String","Latitude":0,"Longitude":0,"Vehicle_Long_Name":"String","Heading":0,"Speed":0,"Route_Name":"String","Block_ID":0,"Status":0,"Status_Text":"String"}]
```

Figure 17 Response DTO definition as a lightweight, serializable JSON file.

Performance benefits from the above definition come in two parts. The DTO request and response makeups (name and payload) dictate to the service where it should be routed. This eliminates the reading of an XML header and reduces the Read/Write time of the request process, making it faster. Network load is also reduced by this slimming of the request DTO since message sizes are smaller. ServiceStack works with HTTP and not against and is built on pure c# in the ASP framework, requiring less overhead and system calls for retrieving and sending packets over the network. Secondly, custom file types that satisfy application requirements or purposes can be provided directly from the HTTP response since ServiceStack does not require any particular file type or header to be a vehicle for metadata.

The business logic and application levels

It is at this stage that student researchers can add in the application specific and use specific logic their applications will use. API methods in their simplest form have the purpose of taking a DTO that encapsulates the parameters needed for a model's stored procedure. The method acts as a delegate to validate the data of the request, approve permissions and authentication levels, and then safely execute the stored procedure on the database. Lastly, it packages the results into a response DTO and returns it to the caller. These method calls can use the benefit of making several stored procedure calls, checking against other results, and performing work that manipulates the data for specific

purposes we were not allowed to perform in the database model. Writing the transit data analysis algorithms, file processing and creation methods, data comparison methods, and other automations of the transit work students perform in the GLaaS API and not just in their applications is the subtle, advanced benefit of this project. Future researchers can now just composite these API calls together to perform more complicated functionality without worrying about both the integrity of the data, the creation of the mechanisms – since the original application they were made for should be tested and proven – and can only have to write code they know they will use. This API model fits with the mantras of the agile philosophy of small, iterative functionality in succession; lends itself to purposeful code and documentation; and localizes maintenance and replication to specific areas of use. The unintentional benefit that was gained was also now a complete knowledge of the data lifecycle is no longer necessary to create new applications, and larger applications can easily be collaborated on by breaking up the work between the strictly defined aspects in this framework.

Authentication and security

Authentication and security implementations were outside the scope of this project, but it must be considered due to the transition of hard access via database authentication (closed pipe) to web reuse (open pipe). Goals of authentication with this API are to localize them to the specific features and requests, and not to the application. This is similar to designing to the data structure and not to the use. Concerns are data spoofing and impersonation of real-time data pretending to be from an RTA it is not.

Traditional method would require a custom encryption system for user and password keys that accompany the DTO to the call. In ServiceStack, authentication is built in as an upfront configuration in the main entry point of the service. Basic authentications like user and pass as well as new Restful authentications such as OAuth[32] are supported. Developers are even free to develop application specific authentications custom for features. While implementing any of these authentication techniques were not performed for this project, the perceived ease of researchers authenticating access to their data in the future contributed to the decision to choose ServiceStack as the API framework.

Results

ServiceStack, SignalR, and a multimodal approach to data services

In previous sections, the different creation rates of transit data structures were discussed. The confusion with these was that most application levels are concerned with the rates at which transit data is consumed. It is now at the API level that the transit data kept separate in the model can be accessed in three different mechanisms similar to how the data was created. ServiceStack is dependent on the pull design pattern like other DTO based web APIs. Techniques are included to try and estimate the push design pattern. In order to best accommodate different consumer purposes, framework examples were created for three distinct conditions.

1. Consumption is specific and more infrequent than production.
2. Consumption is on demand and infrequent and does not approach the rate of production.

[32] A common authentication model, used by the Twitter Rest API among others, containing two sets of public-private encrypted key pairs.

3. Consumption is best done at the rate of production or is frequent.

The first condition is for systems such as GTFS-realtime where data is fetched on a regular basis, but needs to be processed. This is best of static data systems or systems that want to be as loosely coupled to the data and service provider as possible. The second condition is the typical use of data demand that is seen in web and client applications. The third condition is the special case use that relies on new technologies and a special design philosophy. Typically, decoupled services avoid direct connections when communicating with a variable amount of clients. With this API, and real-time created data, the API should enable service to the consumer with the minimal amount of time in between production and consumption. It is not optimal to have to estimate the push design pattern using pull techniques, but to rather have a service feature that simply provides push logic for this data.

Asynchronous file services

This is the solution GLaaS API chose for the first condition. Common data requests of the feature are packaged as files accessible by static URIs. Consumer services can fetch these files on use if they only need to compare against local caches. Static transit data like seasonal schedules in the form of routes, stops and trips and text file databases such as GTFS are perfectly suited for this. Special considerations for this feature include incorporating on demand access to request the generation or refreshing of the static data to best suit the application needs.

Implementation of this aspect of API is currently as a combination of request-response for the file generation, as ServiceStack implementations of features was performed first. The source of these requests is can be a periodic timer based on a regular event like a database change, maintenance to a system or structure change, or an application start or stop. The response is simply a success or failure notification and a URI redirection to where the data feed was saved. Upon finishing this project, no concrete example of this service has been made, but instructions on how to implement GTFS-realtime using this aspect of the API have been written for lab use.

Request-response, on demand services

The primary implementation structure of the GLaaS API is the request-response service implemented in ServiceStack. Each API Feature is contained in an ASP .NET web application project in the API c# solution. This gives us the URI structure of *glaas.geolabvirtualmaps.com/<feature>*. Each project is made of five parts for the request-response portion:

1. Request DTOs as POCOs[33]
2. Response DTOs as POCOs
3. Global.asax entrypoint and configuration
4. Service implementation
5. Webconfig ASP friendly XML

By stating that this is the ideal of what the request-response service structure should have, any other items that implementers create are guaranteed to be for their business logic. Maintainers of the GLaaS

[33] Acronym of *plain old CLR objects* which describes a DTO implemented in .NET (c#, f#, visual basic, etc.)

API engaging in debugging can tell whether the problem is with the request-response structure or the business specific logic by whether the source of the issue originates from one of these five categories or something else, respectively.

Subscription services with SignalR

An ASP .NET adopted library for websocket implementation was researched in an attempt to find a replacement for approximating the pull design pattern. SignalR is a ServiceStack friendly websocket implementation that also is built on REST and allows for dynamic route definition. This allows users wishing to implement real-time consumption of their data to publish a URI relative to their feature URI for client application subscriptions to data without the need to make a request. This would commonly be implemented using a listener for data changes at either the API level, in the cache, or by implementing the observer design pattern on the GLaaS Model. Upon change, the websocket implementation performs the necessary stored procedure or other work only once and pushes the result to all connected subscribers. Currently, websocket is not supported by client libraries other than the most recent versions of web friendly technologies such as jQuery and Java 8. Fortunately, websocket implementations are being made for HTTP clients other than web technologies and scripting languages. In the interim, SignalR has a chain of command to implement fallback methods when websocket fails. Specifically, Long polling, forever frame, and other pull pattern approximations of the push pattern are implemented as those should be supported by all current clients that accept HTTP connections.

Due to this flexibility, SignalR and subscription services implementing the push design pattern can be added to a feature's project by including the library and adding two parts:

1. A hub implementation
2. A startup point

The hub implementation acts similarly to the service in SignalR as the implementation of the business logic and specific functionality. The startup point opens the route URI as an endpoint and also is what is called by the Global.asax (or App initialization if ServiceStack is not needed for the feature).

Discussion

Implementing the API as the last stage of this project was a natural progression from the previous stages. After identifying the issues with existing systems and wanting to cultivate a maintainable and collaborative project that works well with the GeoLab was easy to create after goals and requirements were clearly defined. There are only two specific higher order discussions topics that stand out when making this that are actively disagreed on in the field. The overarching topic of this project was efficiency through verification and validation of data services, and both of these choices – arbitrary to the learning student – have had an incredible effect on those outcomes. These two discussions are written in the effort that future student researchers can benefit from the research into the choice, and can decide on their own what is best for their applications.

REST vs. SOAP

REST and SOAP services have been described in detail related to their implementations in the methodology. SOAP is one of the most widely used data migration mechanisms for mobile and short

term technologies that utilize the clarity and structure XML structures provide. Many client side libraries to automatically create POCOs and DTO implementations from XML web service contracts such as XSDs and WSDLs exist and using web services and APIs implemented in SOAP is the current favorite of the Transit standards leader APTA in their TCIP documentation.

REST is the widely preferred choice among developers when lightweight, data serialization in a timely manner is necessary for performance of the application. Sending images and other media, data streams, and constantly changing data feeds benefits from the compact nature of REST request and response payloads which shortens the turnaround time and benefits both parties. The original complain of REST structures is that the definition of how to use the data is not built into the REST request and response definitions, but as we have seen in our implementation is overcome through the use of generated metadata pages.

The conclusion reached in this project is that the tools do affect the way the implementer creates their applications. The philosophy of this API implementation and the GLaaS framework is to encourage and strictly regulate the approved means so that they do not get contrived such as formally structuring the parts of the API. The other predominating thought is to not discourage the use of competing tools, but only make sure that each tool is used for its best intended purpose. By following these rules, the API structure will be clear, readable, manageable, maintainable, and secure for future development.

CLR Triggers vs. SQL broker

This issue encompasses a broader topic, but the scope of this project was specifically to answer the following question: *What is the best push mechanism between SQL and c#?* Which entity has the responsibility of notifying the other when data changes. Various implementations employ the API to monitor changes in the database. These parts of the API then employ the push pattern to their subscribers. It seems counter intuitive that with sophisticated data management tools such as SQL server, the API cannot have the server notify it of events on its database objects such as tables. Several solutions were tested in creating the SignalR implementation of the GLaaS API, and two competitors for database monitoring came out.

CLR triggers are application level database objects. They are implemented in the server just like the SQL triggers discussed before and suffer the same dependencies and issues those triggers did. The difference is that CLR triggers execute c# code callbacks instead of queries. This would allow the CLR trigger to listen on insert events on database objects, siphon a copy of the inserted data, and supply it to the SignalR hub to be pushed to subscribers. This is a perfectly fine implementation of the observer design pattern, where the CLR triggers are *observing* the GLaaS Model.

An alternative to this implementation was sought to avoid these problems. It is a concern that during the testing phases of subscriber API features issues with trigger logic could throw exceptions. Breaks in these triggers would abort the entire insert query chain that they are attached to. Fitting in to the current framework, it is important the any observer of the GLaaS Model attaches to the insert stored procedure that each feature has. Recently, the ADO .NET framework added support for

SqlDependency. It is a way to get notifications from the SQL broker when specific queries are executed. While this method would not pass the data inserted to the application level like CLR triggers, the query being observed should be enough to know what table and objects have changed. Then, the only added step is performing the select stored procedure that corresponds to the feature. This may not resemble the purest form of the push design pattern, but it is a valid implementation of the pattern since data is request and sent upon change events with only one step in between. The time between data production and consumption is kept to a minimal satisfying the same criteria as push.

Conclusion

Only a limited number of examples of API feature implementations were made under the scope of this project. The APIs framework and protocol are the most important results of this project. Future researchers can maximize reuse and stability of their data and operations through minimal work by adopting this API structure. Acting as the application level counterpart to the GLaaS Model, this API acts as a smartly documented access point for that data. Creating and structuring this API and Model has been a way to implement the abstract structures discovered when performing the analysis and quality assurance measures in the initial parts of this project. This API is the accumulation of the techniques, data definitions, and all of the decisions and methods researched at the Lab. This documentation acts as the explicit expectations of the project, the wiki documentation of the API acts as the practical knowledge that Lab researchers can implement and design against, and the physical examples in the projects code and its auto generated metadata acts as the training implementers and application developers in the lab can use to collaborate with researchers.

Final Statements

This project evolved dramatically over its course of production. Originally a pure analytics project of a data collection, it soon became a quality assessment and reflection project on an existing system. Through the definitions discussed, custom use of data can be easily produced from this protocoled structure. Specific application can make continued use out of the methodologies practiced and explained. Many implementations of techniques are dependent on the technology they are implemented in because of the concurrency between learning and implementing. A protocols form can dictate its function, and vice versa, such as in the nature of WCF and ASP MVC services and their SOAP XML message binding.

From a predominantly structure and definition approach, a concrete product and implementation came from this project. Integration, use, and adoption were outside the timeframe for this project, but future researchers looking to do analytics on the data can utilize this as precedent. By following the structure outlined, future transit data implementations can benefit from either consuming these services directly or modeling their own after these as examples.

The future work of this particular project will be in implementing common GeoLab utilities as features of the GLaaS Framework. These immediately include geospatial wifi analysis for the Cape Cod Region; GTFS creation, automation, and integration as a static data reference; and a light-weight refactoring to the current mapping applications. These implementations can be carried out by future researchers in the lab, and by adhering to the protocols, guidelines, and algorithmic suggestions in this paper, can be successful and contribute to a reusable structure.

Index

A

Agile Software Development 4
Authentication 1, 24, 42
AVL..... 3, 6, 7, 10, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 27, 28, 29, 30, 35, 37, 39, 40, 49

B

block 10
BusLocator .. 3, 4, 14, 15, 16, 17, 19, 21, 27, 36

C

CCRTA 3

D

Differential Privacy 25

E

ESRI 6, 11
ETA..... 3, 4, 6, 10, 11, 12, 15, 16, 17, 18, 19, 21, 35
exponential moving average 21

G

Geo Lab as a Service .... *See* GLaaS
GeoGraphicsLab 4
GeoLabVirtualMaps...................... 3
geolocation 8, 9, 10, 25, 28
GLaaS 1, 5, 14, 18, 21, 25, 26, 27, 30, 31, 33, 34, 36, 37, 38, 40, 42, 43, 44, 45, 46, 47
GPS MDTs.................................... 3
GTFS 12
GTFS-realtime............ 11, 19, 24, 43

N

NextBus 3, 6, 16, 17, 18, 19
NextBusETAHelper 4

P

pull 24
Pull design pattern 19

R

REST 1, 38, 44, 45, 49
route 9
RTA 3

S

ServiceStack . 1, 5, 37, 38, 39, 40, 41, 42, 43, 44
SignalR 1, 5, 42, 44, 45
stop... 9
Stored procedures 31

T

TCIP... 6
Transit data structures 7
triggers 1, 28, 31, 32, 45
trip 9

V

vehicle 9
Vehicle 9

W

websockets 21

Bibliography

Cham, L. *Understanding bus service reliability: A practical framework using AVL/APC data.*Thesis, Master of Science in Transporation, MIT, 2005.

Chan, J. *Rail transit OD matrix estimation and journey time reliability metrics using automated fare data.* Cornell University, 2005.

Dwork, Cynthia. Differential privacy. Microsoft Research, 2013. http://research.microsoft.com/pubs/64346/**dwork**.pdf

Engelbrecht, R., Rector, A., Moser, W. "Verification and validation" *Assessment and evaluation of information technologies in medicine.* 51-53. IOSPress, 1995.

Furth, P., et al. *Uses of archived AVL-APC data to improve transit performance and management: Review and potential.* Transit Cooperative Research Program, June 2003.

Lentfert, P., Overmars, M. *Data structures in a real-time environment.* University of Utrecht: Netherlands, 1988.

Li Li; Wu Chou, "Design and Describe REST API without Violating REST: A Petri Net Based Approach," Web Services (ICWS), 2011 IEEE International Conference pp.508,515, 4-9 July 2011.