Taxonomy of an Application Model: Toward Building Large Scale, Connected Vehicle Applications

Nam Ky Giang
University of British Columbia
Vancouver, BC, Canada

Victor C.M. Leung
University of British Columbia
Vancouver, BC, Canada

Rodger Lea
Lancaster University
Lancaster, United Kingdom

ABSTRACT

With the advent of advanced computing systems beyond personal computing, such as mobile computing, cloud computing or recently, vehicular ad-hoc network, it is crucial that we understand the application development process of each type of these systems. Better understanding of how applications are built in different environment allows us to design better application models and system supports for developers. This paper studies the taxonomy of application models and defines its consisting aspects, namely application scope, application abstraction level, application structure, communication model and programming model. With the better understanding of the application models in general, we lay out the requirements for developing a class of large scale connected vehicle applications.

ACM Reference Format:
Nam Ky Giang, Victor C.M. Leung, and Rodger Lea. 2019. Taxonomy of an Application Model: Toward Building Large Scale, Connected Vehicle Applications. In *9th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications (DIVANet '19), November 25–29, 2019, Miami Beach, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3345838.3356001

1 INTRODUCTION

The process of developing applications has been practised since the beginning of the computing era. As the computing systems evolved, we see more classes of applications being built. For example, there is a transitioning from desktop applications to mobile applications, and to applications that run in large scale data centres. We also see applications that run on small embedded devices, such as smart watches, or personal activity trackers; or applications that run on large scale wireless sensor networks.

Despite the long history of application development for many types of systems, to date, we still see this process happens mostly in an ad-hoc style. That is, to develop applications for a particular class, we design the development process particularly for such a class without connecting them with other classes of applications. This leads to many different ways to develop different kinds of applications, most of them are application-specific, or domain-specific. This process itself, is not reusable as new development processes have to be designed specifically for any new type of applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DIVANet '19, November 25–29, 2019, Miami Beach, FL, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6907-7/19/11...$15.00
https://doi.org/10.1145/3345838.3356001

While there were many programming models designed for such many different types of systems, the notion of application model or a general skeleton that is applicable for all types of applications has not yet been defined. We believe the construction of complex applications does not solely depend on the language aspect (programming model), but also on a range of other criteria, such as application structure, abstraction level or communication model. Certainly, some of these criteria were discussed and surveyed among many existing programming models, it is still valuable to properly define the notion of application model as a skeleton for constructing different types of applications.

To fill this gap and to help us with our target class of large scale, connected vehicle applications, we start designing such general skeleton for any type of applications, which we call the taxonomy of an application model. Our aim is not to have an exhaustive list of items that constitute an application model, but rather, to produce a comprehensive understanding of the application models that can be extendable for any new type of applications.

With the developed taxonomy, we propose a set of requirements for a class of large scale, connected vehicle applications. We envision that, due to the proliferation of cheap computing devices, more and more computing resources will be surrounding us in near future. These resources can include computing devices running on-board a smart, connected vehicle, road-side units or cell towers. These entities are distributed in a large area such as across the whole city, and are highly dynamic because of the mobile nature of vehicles. Such complexities pose great challenges to the application development process as the developers are overwhelmed by the number of participating devices, their heterogeneity and their dynamic nature.

Our main contributions are:

- Through the developed taxonomy, we contribute a better understanding of the construction of applications in general. We show that for most applications, the developers and the users should pay attention to five aspects, namely, *scope, abstraction level, organisation structure, communication model, and programming model.*
- Also based on the taxonomy, we show that, there are unique requirements for developing large scale, connected vehicle applications that are different from many other distributed applications.

It is worth nothing that, our focus in this paper lays at the constructing of an application model in general, where we discuss what constitute an application models and how they affect the development process. While we also demonstrate how our developed taxonomy can be used in designing an application model for connected vehicle applications, a particular solution for this class of

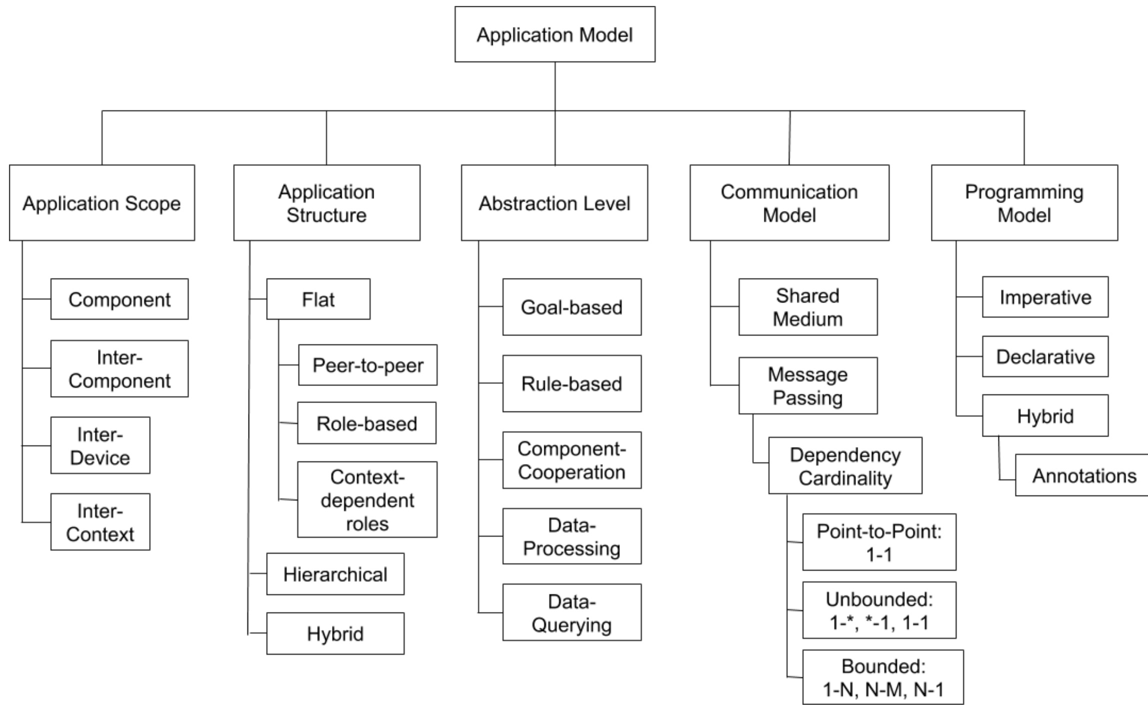

Figure 1: An analysis of various application model

applications is out of scope of this paper and is the topic of our current research.

2 APPLICATION MODEL TAXONOMY

We consider an application model to consist of five aspects, *scope, abstraction level, application structure, programming model and communication model*. This section describes these aspects and their sub components in details. An overview of these aspects is shown in Fig. 1.

2.1 Application Scope

The scope of an application defines the boundary of the application and its external ecosystem. For example, a typical distributed application is that of a traditional Internet application, which usually consists of a client terminal and a server component that communicate with one another. Generally, we can perceive this scenario to be one application that has two interconnecting components, a client and a server. If we "zoom in" each of these components, either the client or the server could also be seen as another utterly independent application. These smaller applications can also be further decomposed. The server component, for instance, could include a web server and a database; the client terminal could include a user interface and a background running process.

Our observation is that different levels of application scope come with different concerns from the developer; consequently, they need different toolings and supports. For example, at the atomic component level (we use the term atomic here to denote that such component *should not* be decomposed further), most of the concerns are how to derive efficient computing algorithms and how to choose the suitable programming primitives and constructs. At a higher level of scope, we start to employ different design patterns to facilitate the inter-component interaction. One of the primary considerations for such a level is whether to use object-oriented or functional programming. Moving higher up the scope levels, we start to think about the communication patterns among components that are distributed across machines.

2.1.1 Component scope. Component scope is the most basic application scope where the developers focus on the implementation of algorithms and basic programming concerns such as the languages, or the programming models (e.g. imperative, declarative or hybrid) to solve the functionality problem of one component.

With this application scope, the main task the developers are doing is purely implementing computation activities.

Examples of component scope applications are sorting, mapping or reading sensor data.

2.1.2 Inter-Component scope. In this application scope, the components themselves are usually well encapsulated and have well-defined interfaces. Often, the interaction among those is object method calls or function invocations. The task of the developers is to combine the components to fulfil the applications' requirements.

Primary concerns in this application scope usually are choosing the right design patterns to maximise the efficiency of inter-component interactions, choosing the right component organisation (e.g. flat vs hierarchical, see later sections), or to learn the components' interfaces.

Examples of applications in this scope are general computer applications, such as desktop apps, mobile apps. That is, most of these applications are developed from many subcomponents, libraries.

2.1.3 *Inter-Device scope.*

In this application scope, the components themselves can be distributed on top of several physical devices. Thus we have applications that run across different devices. Many applications within this scope can also be deployed in a single machine; however, desired features such as load balanced ability, scalability, or availability motivate the distributed deployment of the application's components.

The main concern in this application scope is to coordinate the interaction among the components across devices, for example, to ensure the communication follows some strict orders or rules (e.g. message delivery semantic that is exactly one or at most one).

Examples of inter-device application scope include a traditional client-server application, a load-balanced web server, or a high availability file storage system.

2.1.4 *Inter-Context scope.*

In this application scope, not only the components are deployed in a distributed fashion on top of many devices, the context of each device is different from one another and can interfere with the application's logic. The running devices in this scope are usually heterogeneous rather than homogeneous as in inter-device scope applications. They come from all shapes and sizes and operate in different conditions.

The main concerns of the developers when developing applications at this scope can be how to coordinate the inter-component interactions and how to cater specifically to the physical context of the underlying devices.

Example applications in this scope are Smart City, or Large Scale Connected Vehicle applications, where the physical context (e.g. location, network connection) of a computing device also participates to the application's logic.

2.2 Abstraction Level

We define the abstraction aspect of an application model as the level of flexibility in expressing the application's logic. This definition is not to be confused with the abstraction level of programming models, which is often used to denote the flexibility or transparency when the developers want to tap into low-level system concerns (e.g. working with memory, controlling networking stack). For example, TinyDB [10], a query processor for Wireless Sensor Network (WSN), provides a very high level of abstraction in terms of programming model [9] (it abstracts away the technical details of a WSN). However, with regard to our definition of application's abstraction level, it has the lowest level of abstraction (it provides the most flexible way to express an application logic, i.e. through queries, but the users are also required to master its query language).

Generally, applications with a higher level of abstraction are easier to operate, but functionality is limited. Meanwhile, the ones with a lower level of abstraction are more flexible (i.e. they allow users to do many different tasks, even ones that are not foreseen by the developers) but are more difficult to operate.

We define five abstraction levels, from the highest level of abstraction to the lowest as follows: *Goal-based, Rule-based, Component Cooperation, Data Processing and Data Querying* applications.

2.2.1 *Goal-based.*

Goal-based applications offer the easiest way to interact with the application state and its data, normally through just a push of a button. Most of the applications we see today, such as mobile or desktop apps belong to this level of abstraction.

An example application is the thermostat, which allows the user to key in their desired room temperature. The application itself coordinates its components (e.g. by turning on and off ventilation fans) so that the desired temperature is met. A complete solution to this example is presented in [11] where the application depends on a semantic reasoning process to coordinate the components to achieve the users' predefined goal. Since the application is very easy to use, its functionality is limited to the features that are hard coded by the developers (e.g. we cannot express the requirement such that when temperature drops below 20 degree Celsius for 10 minutes, start the ventilation fan).

2.2.2 *Rule-based.*

Applications at this level of abstraction are slightly more flexible than the previous one and require a certain level of involvement from the users' perspective.

In this level of abstraction, the users are allowed to specify the requirements through a set of rules rather than a specific goal. A set of logic skills is required to make use of the applications. *If This Then That (IFTTT)* [1] is a typical example of this level of abstraction. When the user installs the IFTTT application, they can express their application requirements through a set of simple rules in the form of *if this then that*. For example, if the time is night time, dim the light and reduce the temperature.

2.2.3 *Component Cooperation.*

At this level of abstraction, the components of an application are exposed to the users so that the users can control their cooperation within the application. The applications at this level are much more flexible; consequently, the users can leverage them to achieve many different tasks, sometimes out of the imagination of the developers.

This level of abstraction strikes a balance between the users and the developers' involvement. The developers do have a set of criteria to define the usage of their applications; however, they also give the users the power to change the applications' behaviours the way they want.

Examples of applications in this case are visual programming, such as the Microsoft Flow [2], Max [3] or Node-RED [4].

Since the purpose of these programs are for expert users to construct their applications (applications that make applications), the notion of application developers and users, in this case, becomes system developers and domain experts.

2.2.4 *Data Processing.*

Data processing is the next level down the application abstraction scale, where the main purpose of an application is to modify or transform its data.

At this level, the users are given with some data processing primitives (e.g. min, max, stdev) and should know how to use these primitives for their needs. The detail implementation of these primitives is the responsibility of the developers.

[1] http://ifttt.com
[2] https://flow.microsoft.com/
[3] https://cycling74.com/products/max/
[4] https://nodered.org

Some examples in this scenario are the spreadsheet application or the awk command in Linux operating system. The spreadsheet application allows users to do many processing tasks on their data. It naturally requires the users to be proficient at the tasks they are completing (e.g. using the max function to find a maximum). Awk command in most Linux distributions is a powerful tool for processing text files. It is one example of an end user applications that require specific skills to master the operation of such applications to carry out the user's needs.

2.2.5 Data Querying. Lastly, data query-based applications provide the most flexibility to address the application's requirements. With this level of abstraction, the users have full control over what they want with the application by issuing queries to access and manipulate the application data and state. Examples of this are traditional database systems and their query languages, such as SQL. To illustrate the flexibility, most of the web-based or enterprise applications can be constructed from a single database layer where users can issue queries and manipulate the business data directly from the database. Another example is TinyDB [10], a query processing system for Wireless Sensor Networks that allows access to sensing data and a range of data aggregation operations. TinyDB itself could be seen as an application for WSNs, which allows users to interact with the networks via issuing queries.

Data query-based applications provide the most flexibility for users to express their requirements. However, the users have to master the query language for their needs.

2.3 Application Structure

Unlike the scope, or abstraction level of an application model, the application structure aspect taps into more details about how the developers designed an application. Generally, subcomponents of an application can be organised into either a flat or hierarchical structure. While a hybrid one does exist, as we shall explain, it has inherent drawbacks.

2.3.1 Flat Structure. Flat structures usually need a dedicated coordination mechanism to facilitate the interaction among subcomponents, such as a service registry that holds information about the components. Flat structures can be further categorised into peer to peer and role-based structure. In peer to peer structure, application's components are generally homogeneous, the only difference among them is the data they hold. On the other hand, applications with the role-based structure are more heterogeneous in terms of their sub-component composition. Accordingly, these subcomponents are different from one another and are bound together in one program by dedicated coordination mechanism.

Examples of a flat structure include peer-to-peer applications, such as torrent file sharing, or blockchain-based smart contracts. In these types of applications, special component discovery services are usually required for inter-component interaction. That is, in torrent file sharing applications, it is the torrent trackers that facilitate this interaction, in blockchain-based applications, it is the DNS Seed nodes.

Example applications with role-based structure include micro-service (components play the role of different services), client-server (client components or server components), or map-reduce (mappers and reducers) applications.

2.3.2 Hierarchical Structure. Hierarchical structures can be self-coordinated, meaning the components themselves can discover and interact with one another without a dedicated service.

Examples are domain name systems, applications that rely on MQTT message broker topic naming scheme.

These component structures have their pros and cons and are chosen based on the application's characteristics. An essential aspect of evaluating these two organisational models is communication among components. In flat architecture, the components usually communicate with one another either directly or via a communication broker. In hierarchical architecture, inter-component communication is usually narrowed to parent-child communication. Thus, two components A and B can only communicate if they share a common ancestor component. However, there is no need for a dedicated coordination entity in the hierarchical structure as the components can interact via their direct parent or children.

Hybrid structures are possible and do exist [7] to take advantages of both worlds. However, when the components in a hierarchical structure communicate directly with one another, their parents might have difficulty keeping track of the state of their children.

Applications based on Akka actor system [5] or recently React programming model [6] are examples of the hybrid structure.

2.4 Communication Model

Two popular communication models for applications are message passing and shared memory models [1]. While these models are from communication patterns in distributed systems, it also applies to non-distributed applications. For example, method calls among components within a non-distributed application represent the message passing communication model. Meanwhile, global-variable scope represents the shared memory model.

2.4.1 Message Passing Communication. While message passing is a well-known communication model in many distributed applications, large scale applications at inter-context scope exhibit a different communication cardinality that we do not see before. For example, we have point-to-point (1-1) or point-to-multipoint (1-*, *-1, as in broadcast or aggregate) communication cardinality among the components. Due to the context scope, sometimes it is irrelevant for a component to broadcast itself to the whole ecosystem. In such cases, bounded group communication among a small set of components that share the same context might be necessary. Thus, we have another category of communication cardinality beyond the unbounded one, which is bounded group communication.

2.4.2 Shared Medium Communication. In shared medium communication models, application's subcomponents communicate via a shared data medium where one component writes data to the medium, and the others read from the same location. This is sometimes referred to as implicit rather than explicit communication as communication is a side effect of the actual data sharing process [1]. Shared medium communication is also regarded as the control-flow hidden interaction model [4]. This is because the control flow is not

[5] https://akka.io
[6] https://reactjs.org

visible, nor easily to be grasped as the components do not interact directly with one another.

Examples of shared medium communication models include applications that are based on tuple space or publish/subscribe model.

It is worth noting that the shared medium communication models do not usually require a dedicated coordination entity to facilitate the interaction among components. This is because the shared medium itself becomes the coordination entity that glue together the applications' subcomponents. In contrast, message passing communication models do require a dedicated coordination entity to dictate to where each component should direct the message.

2.5 Programming Model

The last aspect of constructing an application model according to our definition, is its programming model. This is the most involved aspect of the application development process. Generally, researchers categorised programming models into declarative or imperative languages.

Sometimes, a hybrid category is introduced [12] in which, the language to develop application components is imperative while the language to develop the interaction among components is declarative. Surprisingly, if we apply the definition of application scope in this thesis, this view is no longer suitable. That is if we see the scope of the application to be inter-component scope, the components' internal implementations become irrelevant, the application is, therefore, developed with declarative languages only. A similar argument could be applied to the component scope, where the declarative interaction among components is irrelevant.

Another related concept is the notion of node-level and system-level programming, which is borrowed from the WSN research. In programming WSN, node-level programming refers to the practice of developing applications for individual nodes while system-level programming refers to developing the collective sensing system as a whole. Imperative language tends to do well in node-level programming while declarative is more suitable for system-level programming [3].

It is generally perceived that imperative language is usually more suitable for expressing the functional aspect of an application while declarative is used to express the non-functional logic [12] [2]. For example, to execute a sensor reading from one device, imperative language is used to call system procedures and deliver the data packages. However, to specify logic such as execute this sensor reading only from 9 AM to 5 PM every Sunday, it is more concise for the developer to specify this using a annotation approach than an imperative one.

Thus, we define another notion of hybrid programming model to denote a mix of annotations within the imperative or declarative code. Aspect Oriented Programming [5] is an example of hybrid, annotation-based programming model.

3 TOWARD BUILDING LARGE SCALE, CONNECTED VEHICLE APPLICATIONS

This section analyses the defined aspects of application models and gives the recommended requirements for designing an application model for our large scale connected vehicular applications. A summary of our recommendations is highlighted in Fig. 2.

3.1 Application Scope

Due to the large-scale distribution of computing resources, their physical location, or more generally their physical context, becomes an important factor in this class of applications. Thus, we say that our class of applications falls into the inter-context application scope.

For example, in [8], the authors deployed a smart city application where cameras are mounted on city's garbage trucks, and the captured video streams are used to identify the road markers that need to be repainted. In this application, the city might only want to do the road marker classification in specific regions within the city. Since the garbage trucks are mobile entities, the developers have to program the application so that the software components are enabled or disabled appropriately based on the cars' current location. Furthermore, when leveraging video streams from vehicles' onboard cameras to do the image classification, process those video streams is a computation and communication intensive task. It can require the application to exploit a wide range of road-side units or mobile base stations.

In these scenarios, we see the critical role of the physical context of the computing elements to the correctness of the application logic. We also see how the application developers might use it in their applications.

Therefore, to develop such class of applications, we argue that the application model should provide developers with the ability to express their application requirements based on the physical context of the underlying connected vehicle.

At the same time, we also found that there is another separation of concerns when building inter-context scope applications, which are not present when building typical applications. The first concern is how to implement the application logic (component-scope), and the second is how the system can be deployed across different physical contexts (inter-context scope).

The application does not only run on a single vehicle but across cloud servers, road-side units and connected vehicles under various contexts. This involves writing components that encapsulate functionality, can be easily distributed and can communicate with other required components in various ways. The second concern is the need for the inter-context application developer to specify how the system as a whole decides where groups of application components should be split, run, and how do they communicate with each other.

In essence, the developer needs to adopt an inter-context programming mindset, one that involves specifying context-dependent constraints such as location, computing and network environment, replication and cardinality requirements. The system can then use dynamic information from the physical environment such as the quality of network communications, current location, current traffic

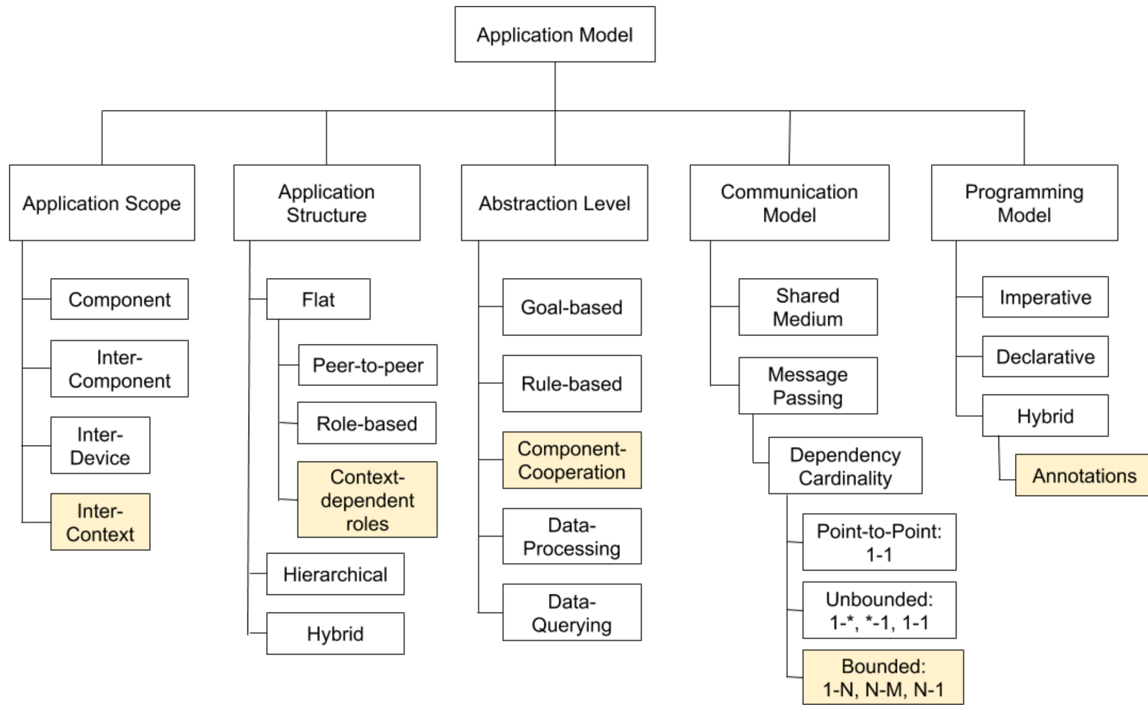

Figure 2: Analysed important design requirements for building large scale, connected vehicle applications

situation and other factors to decide what application components are deployed where.

3.2 Abstraction Level

Due to the large scale of the system, our connected vehicle applications exhibit a substantial level of complexity. Thus, the application model should find a suitable balance between flexibility and expressiveness.

Clearly, we can design an application model at its highest level of abstraction - goal-based - for our applications. For example, define a goal (e.g. average traffic speed in a road segment to be greater than 15 km/h) and all the vehicles, road-side units, and the cloud servers cooperate together to achieve such a goal. The scale of our applications sometimes makes this approach unfeasible.

On the other hand, if we choose to design an application model with the lowest level of abstraction (i.e. data-querying), the cost of implementing any large scale, connected vehicle solution might be prohibitively high.

Meanwhile, the distributed nature of our application class requires an application model that promotes application decomposition capability. This makes some abstraction levels, such as rule-based or data-processing inadequate.

To this end, we found that component cooperation seems to be the appropriate abstraction level for our large scale, connected vehicle applications. As per our analysis, it provides a reasonable balance between flexibility and expressiveness. Naturally, it also fosters the development of large scale distributed applications.

On top of this, an essential requirement to develop an application model based on the component-cooperation level of abstraction is that the communication and computation aspects of one application should be well defined and separated [2].

That is, although our applications, which are distributed system applications in general, can be programmed using many existing models and techniques in distributed systems such as Mobile Agents [17], TupleSpaces [6], RPC-based Middleware such as Jini, CORBA, or RESTful web services. It has been shown that these approaches, which rely mostly on the extensions of the sequential programming paradigm, are ill-suited to meet the challenges of large scale and massively distributed computing systems [14]. The problem of these models and techniques when being applied into such complex applications is that the communication primitives which bind subsystems together are usually mixed within the computation parts.

A typical scenario is when the developers work on the computation code, they have to decide how to carry out the communication at the same time. For example, in Mobile Agent or Tuple Space-based implementations such as LIME [13], it could be to decide the target agent or tuple space to send the data to after finishing a computation activity. In another implementation such as RESTful web service, it could be to decide which API endpoints to invoke on the remote services.

This mixture of communication and computation aspects usually hinders large scale and complex distributed applications to be built due to the lack of modularity and reusability, as well as the need to cope with complex nature of communication (asynchronous, heterogeneous, volatile). In such a setting, the developers should pay attention to the high-level application logic, which is how software components are glued together to solve a problem, rather than to work on each one.

Therefore, to design for the component-cooperation level of application abstraction, communication and computation should be well defined and separated.

3.3 Application Structure

For applications that operate at large scale and more dynamic scenario, hierarchical structure poses some limitations in terms of scalability and flexibility.

This is because, in large scale, connected vehicle scenarios, the components of one application tend to be arbitrarily distributed, their operations are dynamic and more likely unpredictable. Thus, they need a more flexible communication pattern that can be quickly adapt to their dynamic nature. A hierarchical structure depends on the parent-child relationship among components. This means such relationships have to be established first before any interaction can be made. Furthermore, interaction among components always needs to go through their common ancestor, which could be very high up the hierarchy. At large scale deployment with thousands of devices and tens of hierarchical layers, this can quickly become unmanageable.

For our class of applications, we advocate for a flat structure where inter-component interaction is more flexible and fast to adapt to changes (e.g. see also [15]). As discussed in earlier sections, an important drawback of flat structures is that a dedicated coordination layer is required to facilitate this inter-component interaction.

This requirement has several distinguishing challenges. First, the replication of components in a vehicular network is context-dependent. That is, each application component in such environment could be deployed in many participating vehicles, road-side units, yielding many instances or replications of such component. While these component instances are identical in terms of logic or even data, their physical context makes them distinct entities. Second, since each instance of an application component is a distinct entity, the component composition out of these instances is also non interchangeable.

To illustrate this, let us take an example from the popular class of Internet applications. In traditional Internet applications, while there could be multiple instances of each component, they are usually load-balanced and seen as a single entity from other components' point of view (e.g. a web client sees a cluster of load-balanced web servers as a single entry point; a web server sees a load-balanced database cluster as a single database entry point).

In our connected vehicle applications, due to the large scale deployment of the computing elements, there are many possible compositions of the software components' instances. Each composition might serve at a particular location of the city, and each component instance sees one another as an independent peer. Therefore, these compositions are non interchangeable. In essence, they should be non-overlapped (no shared instances between combinations), and the supporting system has to derive as many of these non-overlapped compositions as possible to extend the coverage of the application.

Thus, even though our applications should use a flat structure, the actual application model has to take into account the context-dependent replication of application components and their compositions.

3.4 Communication Model

For large scale connected vehicle applications that span a wide area and consists of independent software components and various devices, shared memory communication model has its inherent limitations.

Firstly, each of the application's components individually has to conform to the shared medium protocol and the shared medium location. This means the implementation of each component is influenced by its deployment environment, which reduces its reusability. Second, the shared medium could quickly become a bottleneck for the inter-component interaction at large scale deployment. Third, the application's control flow is not visible in shared medium model [4], at large scale deployment, this inherently affects the design capability and maintainability of the application model.

Thus, toward our large scale, connected vehicle applications, message passing is recommended as the ultimate communication model [4].

Going further into the message passing communication model, we also found that, large scale applications exhibit a different style of communication cardinality. In particular, due to the context-dependent replication of application components, communication cardinality here tend to be bounded (i.e. 1-N, N-1, N-M) instead of unbound ones (i.e. 1-*, *-1). This is because of the large scale deployment of vehicles and road-side units that makes the system-wide broadcast or aggregation communication irrelevant to many participating entities (e.g. components only care about data sent from other ones that are close to them).

This means there has to be more support for application-level group communication. Coordinating these group communication is then the central requirement of an application platform for connected vehicles. One of the example of such requirement is the application of vehicle platooning [16].

3.5 Programming Model

As we discussed earlier, imperative programming models are more suitable for expressing the functional aspect of the application's components. That is, imperative programming is more suitable for component-scope applications. Meanwhile, declarative programming is more suitable for expressing the overall components' behaviour and their interaction within a larger application. This is due to the concise of declarative language that is suitable for expressing complex inter-component interactions, which otherwise would be difficult to implement and hard to understand using an imperative programming model.

Toward developing large scale connected vehicle applications, we found that the close bonding with the physical world of the computing devices (e.g. vehicles with on-board processors, road-side units) makes their physical context a particular concern that affects the application's logic. By referencing to the aspect-oriented programming model, we can say that in our class applications, the physical context of the underlying devices could act as a cross-cutting concern with regard to the component's functionality.

4 CONCLUSION

In this paper, we developed a taxonomy of application models that consists of five aspects, which we argue are the most common

and important aspects when designing any development process and model for any type of applications. Our general recommendations was not designed to be exhaustive, but has been shown to be extendable with different classes of applications. To support our research into the development of large scale, connected vehicle applications, this taxonomy helps us to narrow down the design space and provides us the skeleton to develop our application platform.

5 ACKNOWLEDGEMENTS

This work has been partially funded by NSERC (IPS application ID 486401) and the EU H2020 BigClout project (Grant Agreement NÂř723139)

REFERENCES

[1] F. Arbab. The IWIM model for coordination of concurrent activities. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1061:34–56, 1996.
[2] F. Arbab. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, (March '98):11–22, 1998.
[3] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23 – 70, 1993.
[4] D. Arellanes and K.-k. Lau. Analysis and Classification of Service Interactions for the Scalability of the Internet of Things. *2018 IEEE International Congress on Internet of Things (ICIOT)*, pages 80–87, 2018.
[5] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, Oct. 2001.
[6] D. Gelernter and A. Bernstein. Distributed communication via global buffer. *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 10–18, 1982.
[7] K. Hong, D. Lillethun, B. Ottenwälder, and B. Koldehofe. Mobile Fog : A Programming Model for Large Scale Applications on the Internet of Things. In *The second ACM SIGCOMM f (MCC '13)*, pages 15–20, 2013.
[8] M. Kawano, T. Yonezawa, T. Tanimura, N. K. Giang, M. Broadbent, R. Lea, and J. Nakazawa. CityFlow: Supporting Spatial-Temporal Edge Computing for Urban Machine Learning Applications. Urb-IoT 2018 - 3rd EAI International Conference on IoT in Urban Space, 2018.
[9] L. Lopes, F. Martins, and J. Barros. Programming Wireless Sensor Networks. *Middleware for Network Eccentric and Mobile Applications*, pages 25–41, 2009.
[10] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.
[11] S. Mayer, E. Wilde, and F. Michahelles. A Connective Fabric for Bridging Internet of Things Silos. In *2015 IEEE International Conference on the Internet of Things*, volume 0, pages 148–154, 2015.
[12] L. Mottola and G. P. Picco. Programming wireless sensor networks: fundamental concepts and state of the art. *ACM Computing Surveys*, 43(3):1–51, 2011.
[13] A. M. Y. L. Murphy, P. Milano, and G.-c. Roman. LIME : A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodology*, 15(3):279–328, 2006.
[14] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46(C):329–400, 1998.
[15] C. Qiu, F. R. Yu, F. Xu, H. Yao, and C. Zhao. Blockchain-based distributed software-defined vehicular networks via deep q-learning. In *Proceedings of the 8th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications*, DIVANet'18, pages 8–14, New York, NY, USA, 2018. ACM.
[16] B. Ribeiro, F. Gonçalves, V. Hapanchak, O. Gama, S. Barros, P. Araújo, A. Costa, M. J. a. Nicolau, B. Dias, J. Macedo, and A. Santos. Plasa - platooning service architecture. In *Proceedings of the 8th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications*, DIVANet'18, pages 80–87, New York, NY, USA, 2018. ACM.
[17] R. Silveira and S. Filho. The Mobile Agents Paradigm. Technical report, 1998.