RACE: Real-time Applications over Cloud-Edge

Badrish Chandramouli[1], Joris Claessens[2], Suman Nath[1], Ivo Santos[2], Wenchao Zhou[3]*
[1]Microsoft Research, Redmond [2]Microsoft Research, Aachen [3]University of Pennsylvania
{badrishc, jorisc, sumann, ivosan}@microsoft.com, wenchaoz@cis.upenn.edu

ABSTRACT

The *Cloud-Edge topology* — where multiple smart edge devices such as phones are connected to one another via the Cloud — is becoming ubiquitous. We demonstrate RACE, a novel framework and system for specifying and efficiently executing distributed real-time applications in the Cloud-Edge topology. RACE uses LINQ for StreamInsight to succinctly express a diverse suite of useful real-time applications. Further, it exploits the processing power of edge devices and the Cloud to partition and execute such queries in a distributed manner. RACE features a novel cost-based optimizer that efficiently finds the optimal placement, minimizing global communication cost while handling multi-level join queries and asymmetric network links.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

Keywords

Cloud, Smartphones, Query, Optimization, Streams

1. INTRODUCTION

The *Cloud-Edge topology* — where multiple smart edge devices such as phones are connected to one another via the Cloud — is becoming ubiquitous. Moreover, many of these edge devices are equipped with sensors producing continuous streams of data such as user's GPS location, speed, current activity, device's battery usage, etc. All these have fueled an increasing interest in *distributed Cloud-Edge applications* that provide various services (e.g., notifications or recommendation to users) based on real-time feeds collected from a large number of edge-devices. An example architecture of such services is show in Figure 1.

In this paper, we focus on a class of Cloud-Edge applications that need to continuously *correlate* or join data from

*Work performed during internship at Microsoft Research.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1247-9/12/05 ...$10.00.

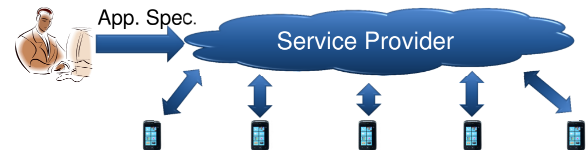


Figure 1: The Cloud-Edge topology.

multiple edge devices. A canonical example is a *friend-finder* application (e.g., *Loopt* app on iPhone, Android, and Windows Phone) that notifies a user whenever any of her friends are near her current location. Enabling this requires correlating real-time locations from users' smartphones as well as slowly changing reference data such as a social network (defining the friend relationship). Other examples of such applications include location-aware coupon services (e.g., GeoQpons on Android and iPhone) that notify a user when she is close to business offering coupons that she might like, mobile multiplayer games (e.g., iMobsters on Android) that need to monitor and react on players' mutual interactions and status, and data-center monitoring applications that provide real-time analytics over resource utilization of servers.

In this paper, we describe RACE, a new framework and system for specifying and efficiently executing distributed real-time Cloud-Edge applications. In RACE, the core application logic of such applications is abstracted as continuous queries running over streaming data from a large number of edge-devices and the Cloud. RACE allows application developers to use LINQ [6] to succinctly express a diverse suite of useful real-time applications. RACE then exploits the processing power of edge devices and the Cloud to partition and execute such queries in a distributed manner.

At the core of RACE is a novel cost-based optimizer that efficiently finds the optimal placement of various operators in the Cloud and various available edge-devices such that the global communication cost (hence the energy overhead of edge-devices) is minimized. Unlike many other previous works on distributed query optimizations (see [3] for a survey), RACE considers optimizing multiple queries together with different data size/rates across devices and possibly asymmetric communication cost. A few recent multi-query optimization works [2, 4] can handle these aspects; compared to them, the optimization algorithm in RACE is several orders of magnitude faster (a few seconds, compared to a few hours) for large numbers of devices, which makes RACE suitable for real-time applications.

RACE achieves its query optimization efficiency by exploiting the specific "star" topology of Could-Edge systems,

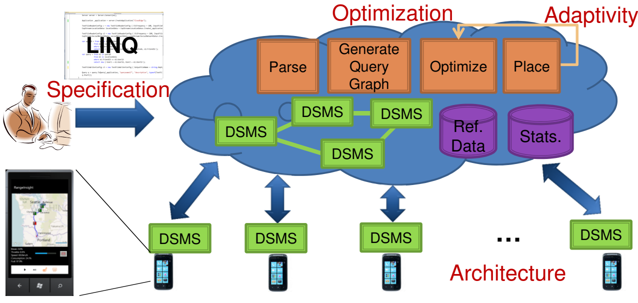


Figure 2: RACE architecture.

shown in Figure 1. We show that for such a topology, locally optimal operator placement solutions obtained by each device independently using a very simple greedy algorithm yields a globally optimal placement solution. This powerful result enables RACE to find a provably optimal placement solution in a very efficient, scalable, and possibly distributed way, without using expensive centralized algorithms such as graph partitioning (as is done in [4]) or linear programming (as is done in [2]).

In the rest of the paper, we describe the architecture of RACE, various use cases, and two concrete applications that we will demonstrate.

2. RACE: SYSTEM DETAILS

Figure 2 shows the overall architecture of RACE. Application developers submit their application logic declaratively using LINQ for StreamInsight. RACE parses the query and generates a larger query graph by materializing slowly changing "reference" data streams (e.g., social network streams) where possible. The optimizer uses collected statistics to find an optimal placement for each node in the query graph.

We have implemented the RACE platform to work with the commercially available Microsoft StreamInsight *data stream management system* (*DSMS*). The RACE optimizer (written in C#) runs in the Cloud and uses a control plane to deploy the generated query fragments and associated metadata such as event types and adapter definitions on StreamInsight engines running at the Cloud end as well on individual edge devices. Our edge application is written using Silverlight for Windows Phone (and hence can be deployed on Windows Phones). A prototype lightweight version of the StreamInsight engine runs inside the Silverlight client application on phones, and executes query fragments deployed by RACE. Events flow between devices via the Cloud using a separate data plane. After placement, future reoptimization may be triggered at the server in order to modify the optimal placement based on current statistics.

2.1 Application Specification

We observe that Cloud-Edge applications are fundamentally temporal in nature — they operate over temporal data (e.g., locations with timestamps) and the application logic is also usually temporal (e.g., notify me if my friends visited this location in the last 5 minutes). Thus, RACE uses a temporal language, specifically LINQ for StreamInsight, to express application logic. For example, the friend-finder query above can be written using a two-way temporal join query in LINQ, as shown in Figure 3.

The final output is a stream of friend pairs `(User1, User2)`

```
var query0 = from e1 in location
    from e2 in socialNetwork
    where e1.UserId==e2.UserId
    select new { e1.UserId, e1.Latitude,
                 e1.Longitude, e2.FriendId };
var query1 = from e1 in query0
    from e2 in location
    where e1.FriendId == e2.UserId &&
          Distance(e1.Latitude, e1.Longitude,
           e2.Latitude, e2.Longitude) < THRESHOLD
    select new { User1 = e1.UserId, User2 = e2.UserId };
```

Figure 3: Specification of friend-finder application.

who are close to one another at any given point of time. LINQ is convenient and can be used either using a standalone GUI (LINQPad [5]) or Visual Studio. The query specification above defines the high-level logic of the query as temporal joins, and references the schemas of the `location` stream and `socialNetwork` stream in a network-topology-agnostic manner.

As another example, suppose we want to find friends who visited our location (say restaurant) within the last week. To specify this, we simply replace the `location` input in `query1` with `location.AlterEventDuration(TimeSpan.FromDays(7))`. This extends the "lifetime" of `location` events to 7 days, allowing the join to consider events from friends within the last week. Our specification is declarative, succinct, and flexible, and can handle a broad range of target applications (Section 4.1 has more examples).

2.2 From Specification to Query Graph

Given a query, RACE analyzes the input stream characteristics and query predicates in order to generate a *query graph*, that consists of stream operators connected together by queues. Continuing the friend-finder example, consider a simple stream representing a social network with three users (A, B, and C). The social network is shown in Figure 4, while Figure 5 shows the corresponding expanded query plan obtained from the original query specification using query rewriting rules. Note that with millions of edges in the social network, a query graph could be very large in practice.

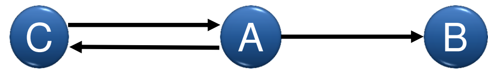


Figure 4: Social network with three users.

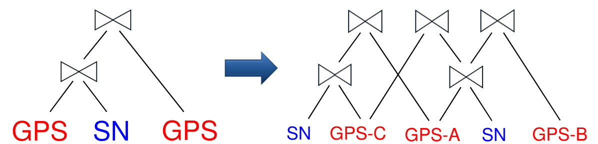


Figure 5: Query plan for friend-finder.

2.3 Optimal Operator Placement

Every edge in the query graph is associated with an *event rate* (in events/sec) for the data stream between the corresponding source and destination operators. Given such a query graph, RACE decides where to place each each operator, with a goal of minimizing total network traffic. The simple scheme of placing all operators at the Cloud, while

simple, can be very inefficient. For general communication graphs, operator placement is NP-Hard [2, 4]. Previous work has shown how to address this in polynomial time for tree-like communication networks (by using a min-cut algorithm) [4]. Our experimental results show that existing algorithms do not scale to graph sizes observed in Cloud-Edge apps. For example, for a friend-finder app with $20,000$ users, the min-cut based algorithm in [4] takes 4.75 hours. This is clearly impractical for (near) real-time applications with dynamic data rates and churn.

RACE solves this issue by exploiting the *star topology* underlying Cloud-Edge apps (see Figure 1). In such a topology, each edge device is connected to the Cloud only. Thus, communication between two edge devices happen through the Cloud. This reflects the dominant form of communication between mobile-phones today. A major research contribution of the RACE project is the development of new algorithms to efficently find the optimal placement for a large query graph. The core intuition is that we can leverage the special Cloud-Edge topology in order to find the globally optimal placement efficiently (the above-mentioned example with $20,000$ users takes 3.2 seconds to find the optimal placement). We refer interested readers to our research paper [1] for the technical details, and instead give a brief example below based on the friend-finder application.

Consider the join query graph for the friend-finder application (see Figure 5) and assume for simplicity that each user has a fixed location stream rate with respect to all their friends (we also ignore the social network streams for simplicity). Instead of trying to place every join operator, we instead consider the alternative problem of deciding whether to *upload* (U) or *not-upload* (NU) each edge data stream. We process each network link between a user (say A) and the Cloud, and compute the optimal U/NU decision for user A and his friends (B and C). It can be seen that the local decision for A made when considering the network link (A, cloud) will be compatible with the local decision made for A when considering the network links (B, cloud) and (C, cloud). Thus, we can conclude that such the U/NU assignment is globally optimal (it is easy to derive the operator placement from a given set of U/NU decisions).

Interestingly, the above result holds even for practical instances of asymmetric networks with $d \geq u$, where d and u represent the per-message download and upload costs respectively. Further, for $d > u/2$, we can show that the optimal U/NU decisions can still be found in linear time even if the local decisions are not compatible. Further, we find that the unrealistically skewed case with $d = 0$ is provably NP-Hard; see [1] for details. Our techniques can find the optimal placement for multi-way join queries while supporting asymmetric networks, sharing of intermediate results, as well as cases where data sampling rates for a source stream are different for each join operator with that stream as input (e.g., when we need to correlate location readings at different rates with friends at varying distances). Further, we can handle general query graphs of black-box operators.

Our optimal operator placement strategies are orders-of-magnitude more efficient than current approaches for optimal operator placement, such as those proposed by Li et al. [4] and Kalyvianaki et al. [2]. This allows us to perform periodic reoptimization during runtime, adjusting query locations to optimize costs based on current data rates and statistics. Experiments with real data show that finding the optimal placement is important — the optimal placement for friend-finder on a real dataset based on a social network sample of more than 900K users from Facebook in association with real GPS trace data shows that our placements can be up to 7× better than uploading all data to the Cloud.

3. RACE APPLICATIONS

We now give a few example applications that can be expressed as continuous queries in RACE. We demonstrate the first two of these applications.

► **Pay-as-you-drive Insurance.** This application runs on a smartphone and monitors a user's driving behavior by accessing relevant car data (e.g., speed, acceleration, and braking) from his car's dashboard via Bluetooth. The data is then continuously joined with reference data such as road traffic, weather conditions, and driving behavior of near-by drivers to determine whether the user is a safe or an aggressive driver. The information may be used to determine the user's insurance premium (e.g., he gets a discount if he is a safe driver) and to alert him if he is driving too aggressively. The reference data may come either from the Cloud or from one or more roadside sensors.

► **Friend-Finder.** As mentioned earlier, this application notifies a user whenever any of his friends is near his location. The application finds all the user pairs `(User1, User2)` that satisfy the conditions: 1) `User2` is a friend of `User1` and 2) the two users have been geographically close to each other within the last 5 minutes. There are two input sources to the query representing the core application logic, namely the GPS location streams reported by the edge devices, and the social network data. The GPS locations are actively collected at runtime, whereas the social network data is relatively stable and is readily available at the Cloud.

► **Geo-Coupons.** This application delivers coupons to a user's mobile phone whenever he is within a short distance (e.g., one mile) of a business providing coupons. The set of business and coupons can be selected based on the user's or his friends' interests and recent activities. For example, the user may receive a coupon for a theater playing a movie that his friends have watched in the previous week. The query capturing the core logic of the application runs on these data sources: users' location streams, social network, and users' interests and activities.

► **Taxi Fleet Management.** This application runs over a number of edge devices representing taxis in a city and provides useful analytics (e.g., number of available taxis in an area) to the taxi company as well as to other taxis. For example, the taxi company or individual taxis can see (in real-time) how many taxis are available in various parts of the city. The application also serves customers. A customer looking for a taxi can see how many available taxis are around his location. Taxis also see the count and location of customers looking for taxis.

4. DEMO WALKTHROUGH

We demonstrate various aspects of RACE with the first two applications mentioned in the previous section.

4.1 Application Specification

We show how to specify the applications with continuous queries on RACE. The pay-as-you-drive insurance applica-

```
var query0 = from j in carData
    from r in roadSensorsAndCarConnections
    where j.CarId == r.CarId
    select new
    { CarId = j.CarId, Jolt = j.Jolt, SensorId = r.SensorId };

var query1 = from cs in query0
    from t in trafficData
    where cs.SensorId == t.SensorId
    select new {cs.CarId, DriverIndex =
             F(cs.Jolt, t.TrafficCondition)};
```

Figure 6: Specification of pay-as-you-drive insurance application.

Figure 7: Car simulator used in the demo.

tion can be specified as follows (Figure 6): `query0` joins car's driving information and a database of sensors to determine the jolt (or jerk, the rate of change of acceleration) of the car and the id of the sensor providing reference data for the car. `query1` joins the output of `query0` with sensor data to determine the driver's safety index as a function F of jolt and traffic condition. Note that the query can be trivially modified to compute safety index based on other driving parameters (such as speed) and sensor data (such as average speed of near-by drivers). The query for the friend-finder application is shown earlier in Figure 3 (Section 2.1).

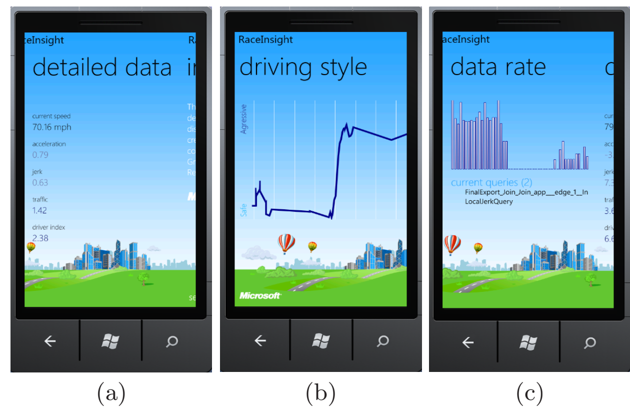
4.2 Input Data

To make both the applications realistic in our demonstration, we provide continuous location streams of a large number of edge devices to simulate users' movement. In addition, we provide one car simulator that users can interact with and control various movement parameters of one edge device (Figure 7). These parameters may affect the optimal execution plan, which we describe next.

4.3 Optimal Execution

We demonstrate the effectiveness of RACE's placement algorithms with three progressively complicated scenarios: (1) the pay-as-you-drive insurance application, with all computation being done at the Cloud; (2) the pay-as-you-drive application, with RACE's optimized operator placement; and (3) the friend-finder application, with RACE's optimized operator placement. Transitioning from (1) to (2) demonstrates the power of RACE's placement algorithm (which may push computation to edge devices). Transitioning from (2) to (3) shows a more complex placement decision—in (2), each user's data is joined with only one sensor, while in (3) each user's data is joined with many friends.

We provide a client application running on the Windows Phone platform and a server dashboard. The client application delivers the final outcome of the queries to the user. For



(a) (b) (c)

Figure 8: Windows Phone client for the pay-as-you-drive application.

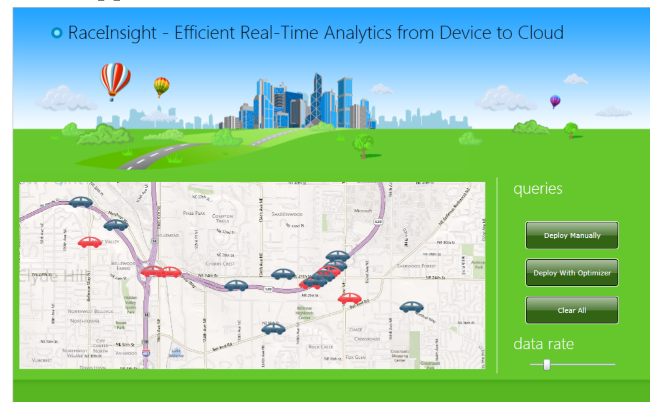


Figure 9: Server-side dashboard for RACE.

the pay-as-you-drive application, it shows the current raw data collected from the device (Figure 7(a)) and the global `DriverIndex` of the user (Figure 7(b)). For the friend-finder application, it will show a list of near-by friends. It will also show the queries running on the device and their bandwidth consumption on the device (Figure 7(c)).

The server-side dashboard shows the locations of various cars or users on a map and statistics such as the overall network bandwidth consumption of the application (with and without RACE optimizations). Figure 9 shows a screenshot.

Acknowledgements. We would like to thank Louis Latour, Eldar Akchurin, Marcel Tilly, and Ilker Dogan for their help in building the demo and providing valuable feedback.

5. REFERENCES

[1] B. Chandramouli, S. Nath, and W. Zhou. Efficient declarative support for distributed apps over smart devices. Technical report, Microsoft Research.

[2] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. SQPR: Stream query planning with reuse. In *Proc. ICDE*, 2011.

[3] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.

[4] J. Li, A. Deshpande, and S. Khuller. Minimizing communication cost in distributed multi-query processing. In *Proc. ICDE*, 2009.

[5] LINQPad. `http://linqpad.net/`.

[6] The LINQ Project. `http://tinyurl.com/avs7wo`.