A Framework for Dynamic Inter-device Task Dispatch with Eventual Consistency

Jihyeok Park
KAIST
South Korea

Joonyoung Park
KAIST
South Korea

Yoonkyong Lee
Samsung Electronics
South Korea

Chul-Joo Kim
Samsung Electronics
South Korea

Byoungoh Kim
Samsung Electronics
South Korea

Sukyoung Ryu
KAIST
South Korea

ABSTRACT

The Internet of Things (IoT) allows various *things* like mobile devices and electronic appliances to communicate over network. *Inter-device apps* can share data between devices and dispatch specific *tasks* to other devices to utilize their resources. The prevalence of JavaScript web apps that can run anywhere providing any browsers opens the gate to unanticipated interactions between devices. However, the current techniques require developers construct tasks to dispatch statically with strong consistency, and they do not provide any disciplined way to develop inter-device apps. In this paper, we propose IDTD (Inter-Device Task Dispatch), a framework that allows developers to construct and dispatch tasks into multiple devices *dynamically* with *eventual consistency* in a *systematic* manner. We provide a high-level architecture of IDTD, prove the soundness and eventual consistency of the framework, and present its practical usability.

CCS CONCEPTS

• **Software and its engineering → Application specific development environments**;

KEYWORDS

Eventual consistency, task dispatch, web applications, JavaScript

ACM Reference Format:
Jihyeok Park, Joonyoung Park, Yoonkyong Lee, Chul-Joo Kim, Byoungoh Kim, and Sukyoung Ryu. 2018. A Framework for Dynamic Inter-device Task Dispatch with Eventual Consistency. In *2018 : 2nd International Conference on the Art, Science, and Engineering of Programming 2018, April 9–12, 2018, Nice, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3191697.3191732

1 INTRODUCTION

IoT has come with various devices that communicate and transfer data over network. Not only devices with modest computing powers such as smartphones but also home appliances like vacuum cleaners may interact with nearby devices. Such interactions between multiple devices are performed by *inter-device apps*. An inter-device app often starts running on a source device, and sends data or dispatches tasks to target devices to use their resources during evaluation.

Current techniques to build inter-device apps belong to two approaches: "task offloading" [14, 23] and "mirroring" [2]. The task offloading technique splits a main computation task to sub-tasks, sends them to other devices that compute them using their own computational power without communication with other devices, and receives the results from the devices to finish the main task. The mirroring technique does not dispatch tasks but sends the screen of a main device to other devices to mirror the tasks of the main device.

However, both approaches are not yet ready for the recent breakthroughs in IoT programming. Even though they are quite simple and perform effectively for traditional programs written in C and Java, they are not suited for IoT programs empowered by the dynamic features and unlimited portability of JavaScript. First, because they should identify and construct tasks to dispatch *statically*, they limit the range of interactions between devices. Moreover, while JavaScript provides more expressiveness and functionalities than C and Java, both approaches do not take such advantages. Second, their simple communication strategies have *strong consistency*, which may be too restrictive for inter-device apps. While the strong consistency guarantees that replicated data in different devices have coherent views, providing it in distributed systems that require high responsiveness like IoT environments is extremely expensive. Third, there is little support for development of inter-device apps. Developers implement them in an *ad-hoc, labor-intensive, and error-prone manner*. No tools nor theories guide how to build inter-device apps. Developers implement multiple apps of the same functionality for various platforms, and end-users should install the same functionality multiple times for different devices.

To alleviate this problem, we propose a new framework, IDTD (Inter-Device Task Dispatch), which allows developers to build inter-device apps that can construct tasks and dispatch them *dynamically* with *eventual consistency* in a *systematic manner*:

- Unlike the current approaches, our framework enables developers to dynamically generate and send tasks using higher-order functions in JavaScript.
- To guarantee eventual consistency between devices running inter-device apps, the framework provides a *shared slicing*

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.
<Programming'18> Companion, April 9–12, 2018, Nice, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3191732

technique. The technique splits a given app into two standalone subprograms that run on different devices communicating with each other. It constructs a task to dispatch in a way that the task can share necessary data of the residual code in the source device from the target device, and enables communication between devices while preserving eventual consistency. To dispatch a sliced task to a target device and run the task on it, the framework should enable the target device to construct a web app from the task and to execute it while communicating with the source device over network. This capability empowers the source device to use resources of target devices as its own ones. The shared slicing technique makes it possible to build efficient inter-device apps correctly.

- The framework provides a high-level, declarative mechanism to build inter-device apps without considering low-level implementation details of how to communicate between multiple devices. When a user selects a task to send from a running web app, the framework slices necessary information from the source device automatically and sends it to a target device. The target device, then, evaluates the task using its own resources such as computational power, screen, various sensors, and private data.

The contributions of this paper are as follows:

- **IDTD Framework.** This is the first proposal for a framework to build apps that dispatch tasks to different devices dynamically while preserving eventual consistency.
- **Formalization.** We are the first to apply eventual consistency to inter-device app development with formal proofs that it guarantees the soundness and eventual consistency.
- **Experiments.** We developed a prototype implementation and made demos of inter-device apps open to the public.

2 EXAMPLE SCENARIOS

Our framework simplifies the development and deployment of web apps that utilize multiple devices seamlessly. For end-users, it eases the deployment of such apps by freeing them from multiple installation. For developers, it mitigates the complexity of developing inter-device apps.

2.1 Easier Deployment

Devices like simple sensor-based ones (e.g., smart bulb) and wearable devices provide limited user interfaces, which often require users install separate controller apps on different but connectable devices. To use such devices, end-users have to make sure that they have installed and maintained the correct version of the controller app on other devices.

Our framework frees end-users from such cumbersome work by dispatching a control task from a to-be-controlled device to a controller device. It assures that the correct control task version is always dispatched. So, users can use any devices as a controller without any additional installation. For example, while the current approaches require users install a controller app for a vacuum cleaner on multiple devices separately, our framework enables the vacuum cleaner to dispatch the controller task to a suitable user device.

Table 1: IDTD supports task construction and dispatch dynamically.

Approaches	Task to dispatch	Task construction	Resources to use
Task offloading	closed term	static	static
Mirroring	rendered data	static	static
IDTD	function closures	dynamic	dynamic

2.2 Simplified Development

Developing inter-device apps consists of several sophisticated steps. First, developers should decide what resources to use from other devices in advance, which determines what techniques to use. Then, they have to extract some code that can be executed on other devices and relevant data to share between devices. To communicate between devices, they should consider the network layer and implement suitable communication protocols, which may not be trivial to most developers. Finally, they should develop several apps—one for each device—to support various devices.

The IDTD framework simplifies the development in many ways. First, it provides a unified way to use different resources from other devices simultaneously in a flexible manner. Instead of contemplating *how* to extract and communicate code and data, developers can focus on the main logic of *what* to extract utilizing the framework. Developers do not need to consider specific devices in advance but simply implement a normal web app that runs on a device; they only need to specify *tasks* to dispatch to other devices with specific *resources* without implementing multiple apps and communication protocols. Then, the framework automatically converts such normal web apps into the ones that can dispatch tasks to other devices.

3 INTER-DEVICE TASK DISPATCH

We propose IDTD, a framework that supports dynamic construction of tasks as function closures. It provides a systematic way to build IDTD apps with eventual consistency. While the framework supports task dispatch between two devices, it can be extended to support more than two devices.

3.1 Dynamic Inter-Device Task Dispatch

As Table 1 summarizes, traditional approaches construct and slice tasks statically at development time, and they determine which resources to use statically as well. The task offloading technique [14, 23] constructs tasks as closed terms without sharing data between the source and target devices. The mirroring technique [2] dispatches rendered data to mirror them in target devices.

On the contrary, IDTD extends the realm of tasks to dispatch by constructing tasks dynamically. Because JavaScript supports higher-order functions, IDTD slices and constructs tasks as JavaScript function closures that may capture data to share between the source and target devices. It constructs tasks and determines which resources to use dynamically.

3.2 IDTD Framework

The framework consists of two parts: 1) it helps developers by converting normal web apps into IDTD apps, and 2) it helps users to execute them easily at run time.

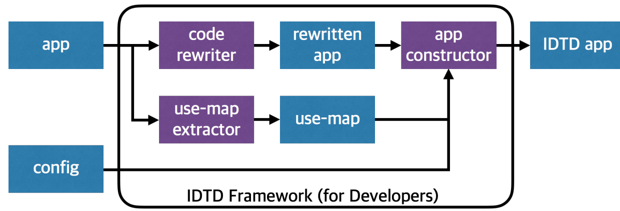

Figure 1: IDTD at development time

Developers specify what tasks to dispatch, and IDTD builds IDTD apps automatically as in Figure 1. It takes two inputs:

- app: a normal web app that its developer intends to convert into an IDTD app
- config: configuration that specifies what tasks to construct in what conditions, and resources from what devices to use for each task

To construct tasks dynamically and to determine resources to use dynamically, the framework takes config that specifies run-time conditions to trigger task construction and dispatch. It may specify to construct a task when a specific device button gets pressed, or when a user invokes a specific event function. In addition, it may specify that a given task uses the screen of a source device and the geographic location of a target device.

Using two inputs, the framework produces an IDTD app. It rewrites app to rewritten app via code rewriter so that rewritten app can dynamically access values originally inaccessible from app such as run-time values of captured variables and event functions attached to DOM objects. The code rewriter rewrites functions like `addEventListener` that add or modify event handlers so that they can record event handler information; it also rewrites functions to record their captured variables. The framework extracts use-map from the given app by using an off-the-shelf static analyzer [13, 21, 24], use-map extractor, so that rewritten app can utilize the information at run time. For each function, use-map extractor collects the following information in the function body:

- global variables
- names and scopes of captured variables
- DOM APIs

Because the IDTD framework is parameterized by use-map extractor, the quality of use-map depends on that of use-map extractor. Then, app constructor builds an IDTD app with rewritten app, use-map, and config.

Users can run IDTD apps using the IDTD framework as shown in Figure 2. As long as devices have the IDTD framework, users can run any new IDTD apps exchanging tasks seamlessly without extra installation using independent techniques: task dispatch and task communication. The framework can construct a "task" from a running IDTD app on a source device and dispatch the task to a target device. Then, the source device keeps running a remaining subprogram, and the target device constructs a subprogram from the dispatched task and runs it. While subprograms are running on different devices, they can communicate with each other.

3.2.1 Task Dispatch. When a user triggers task dispatch via user input during evaluation of an IDTD app, the framework collects

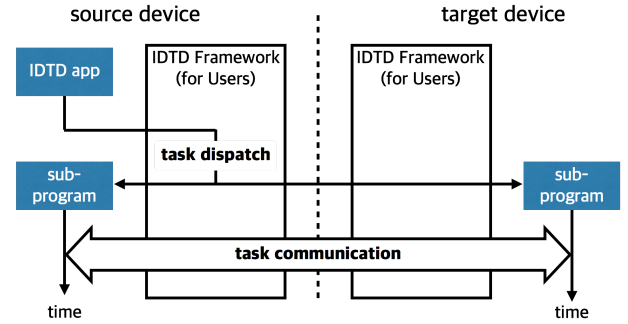

Figure 2: IDTD at run time

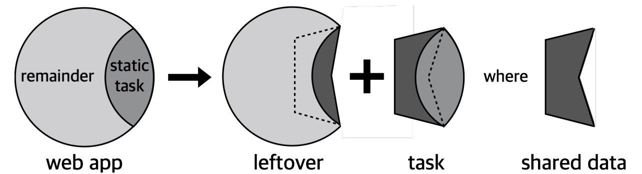

Figure 3: Shared slicing

necessary information to extract an intended task according to config. Then, it constructs a task to dispatch and a leftover, and assigns resources to the task and the leftover according to config. Finally, the framework converts leftover to an executable subprogram. When a task is constructed, the IDTD framework in the source dispatches the task to the target. Then, the IDTD framework in the target converts the task to an executable subprogram. Now that both devices have their own subprograms, each subprogram runs on each device while keeping each other's device information for future communication during evaluation.

3.2.2 Task Communication. While subprograms are running on different devices, they may update shared data on their own devices and send the updates to other devices to reflect the changes. When a subprogram makes an update on a device, IDTD in the device constructs a message with necessary information to make the update on the other device, and sends it to an appropriate device. Then, IDTD receives the message and makes the corresponding update.

3.3 Eventual Consistency

The framework may support various techniques to slice tasks to dispatch, and we present a shared slicing technique among them. To maintain task sizes modest, the technique constructs tasks that share necessary data of the remainder code in the source device using static analysis results. Figure 3 illustrates how the technique constructs a task and a leftover from a running web app. For a given static task, it collects shared data and constructs a task and a leftover; the task is the union of the static task and the shared data, and the leftover is the union of the remainder and the shared data.

While subprograms are running on the source and the target devices, they may update shared data on their own devices, which may be viewed as replicated data in the distributed algorithms

community. The most widely studied consistency between replicated data is *strong consistency* [15], which guarantees the same status over replicated data all the time, but it inherently suffers from performance issues due to the synchronization bottleneck. An alternative approach is eventual consistency, which guarantees the same status over replicated data when no more updates are left.

When a subprogram constructed by the framework updates shared data in its device, it sends the message to the other devices and proceeds its computation without waiting for the message to be reflected in the other devices. Such eventual consistency is surely weaker than strong consistency in the sense that the shared data may be in a status which is not possible in the original web app. However, as the TAO data store of Facebook [12] and the Manhattan system of Twitter [18] witness, eventual consistency is necessary for some classes of web apps, which should choose availability over consistency for better user experience [3].

To formally specify the task communication of the shared slicing technique and its eventual consistency property, we use the formalization mechanism of Burckhardt *et al.* [6]. In their mechanism, an *object* has a (replicated data) type, and the type determines values of the object and two kinds of operations on them: an update operation that affects the value of an object, and a query operation that does not affect the object's value. A *session* contains copies of all the objects, and whenever it has an update operation on an object, it propagates the message to other sessions so that they can also update the corresponding object. An *action* denotes an operation on an object in a session with a unique id, and all the actions in a session are in a total order called a *session order*. Finally, a *replicated data type specification* declaratively describes the semantics of each type's operations using the *visibility relation* and the *arbitration relation* between objects, where the visibility relation denotes whether an action is visible to another action, and the arbitration relation denotes the total order between all the actions.

We describe the shared slicing technique in terms of the terminology of [6]. We consider a device as a session because a device contains copies of shared data, and we consider each component of shared data as an object.

In addition, we use two different replicated data types; Last-Writer-Wins Register (LWW-Register) and Observed-Removed Map (OR-Map). A register type has a memory cell with two operations; the *assign* operation stores some value into the cell and the *value* operation reads the value from it. LWW-Register proposed in [25] is a register type with an arbitrary global timestamp and only the update with highest timestamp wins. In the framework, global and captured variables have this replicated data type. A map type has a map structure with three operations: the *lookup* operation reads the corresponding value of a given key value, the *write* operation stores a given value into a given key value, and the *remove* operation deletes a given key from the map structure. We define OR-Map type adapted from Observed-Removed Set (OR-Set) type proposed in [25], which also uses timestamps to decide which update wins. It represents the object relations, objects, and DOM elements in the framework.

Moreover, since JavaScript event functions are atomic, the timestamp used in the algorithms is not the general wall-clock time but it considers characteristics of JavaScript events: for an action a, $time_{ts}(a)$ is a pair of the time when the JavaScript event that issued the operation of a starts and the time when the operation of a is performed[1]. For comparison of timestamps $time_{ts}(a) < time_{ts}(b)$, we compare their JavaScript event start time first, and compare their operation time when they have the same JavaScript event. Because we can consider top-level code as an event function that starts the program, and all the other operations are evaluated via event function calls in JavaScript web apps, every operation is performed within a JavaScript event.

Both the session order so and the visibility relation vis are defined as in the literature [6]. We define the arbitration relation ar using the timestamp $time_{ts}$ defined above to describe the JavaScript web app semantics more precisely.

4 EVALUATION

In this section, we evaluate the IDTD framework in two respects: we prove that it guarantees the soundness and eventual consistency and we show its practicality.

4.1 Properties of the IDTD Framework

Now, we show that a dispatched task in a target device and a leftover in a source device execute normally without any problems due to the slicing from the original web app. We also show that they will have the same shared data on each device when they deliver all the update messages.

4.1.1 Soundness. We say that a dispatched task in a target device and its corresponding leftover in a source device are *sound* when they do not refer to the parts that are not shared but available only in the other device. Thus, evaluating them does not get stuck because of missing references due to the slicing. To construct tasks and leftovers soundly, the shared slicing technique collects all the used components from the initial static task transitively using sound static analysis results. Note that two cases ask for extra consideration:

- when used components are not accessible dynamically, and
- when used components are determined dynamically.

Examples of the first case are event handler functions attached to DOM objects and captured variables in function closures. For such cases, the IDTD framework rewrites corresponding code to be exposed at source-level so that they are accessible at run time. The second case is function closures whose used components depend on their actual arguments. For this case, the framework uses a sound static analyzer to estimate use-map, a set of all possible used components in the function bodies regardless of actual arguments at run time. Thus, as long as the static analyzer used to build use-map is sound, dispatched tasks and their corresponding leftovers constructed by using use-map are also sound.

4.1.2 Eventual Consistency. The IDTD framework using the shared slicing technique provides eventual consistency. To prove the eventual consistency as defined by Burckhardt *et al.* [6], we need to show the following six axioms:

- Well-formedness axioms:
  - SOWF so is the union of transitive, irreflexive and total orders on actions by each session.

[1] In this paper, time is computed using Lamport timestamps [15].

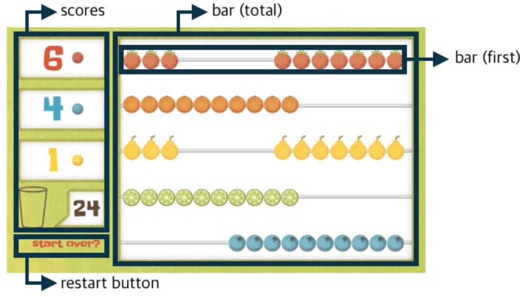

Figure 4: Static tasks for the Counting Beads app

- VISWF $\forall a, b.\ a \xrightarrow{\text{vis}} b \Longrightarrow obj(a) = obj(b)$
- ARWF $\forall a, b.\ a \xrightarrow{\text{ar}} b \Longrightarrow obj(a) = obj(b)$,
  ar is transitive and irreflexive, and
  ar $|_{\text{vis}^{-1}(a)}$ is a total order for all $a \in A$, where ar $|_{\text{vis}^{-1}(a)}$ denotes a set of actions visible from a.

* Data type axiom:
  - RVAL $\forall a \in A.\ rval(a) = \mathcal{F}_\tau(op(a), V, \text{vis}|_V, \text{ar}|_V)$
    where τ is the type of a and $V = \text{vis}^{-1}(a)$.
    For each action, its result should be the same as the result of the specification function of the corresponding replicated data type.
* Basic eventual consistency axioms:
  - EVENTUAL
    $\forall a \in A.\ \neg(\exists$ infinitely many $b \in A$
    $$\text{s.t. } obj(a) = obj(b) \ \wedge\ \neg(a \xrightarrow{\text{vis}} b))$$
  - THINAIR
    so $\cup$ vis is acyclic.

Three well-formedness axioms denote that the relations are well defined. Because we used the same definitions for so and vis from the literature, we proved only the arbitration relation case. The key axiom is RVAL: because the specifications of the operations do not use any session information, even when sessions have actions in different orders, if all sessions have the same visible action set, they all return the same value. We also proved RVAL to show that our implementation returns the same values with the specifications of replicated data types. Among two basic eventual consistency axioms, the EVENTUAL axiom states that, on an object, there exist only finitely many actions that a given action is not visible to. This axiom is satisfied because all the updates will eventually get delivered to other devices as long as their network connections are live. Finally, the THINAIR axiom states that a chain of session orders and visibility relations should be acyclic. This axiom is trivial because timestamps always get bigger, which cannot make any cycles. Due to the space limitation, we refer the interested readers to a companion report [22] for their formal definitions and proofs.

4.2 Practicality of the IDTD Framework

To evaluate the practical usability of IDTD, we selected 5 apps from 20 open-source web apps [11]: Counting Beads, Hang On Man, Make a Monster, Mancala, and Run Rabbit Run. While the framework is applicable to all 20 web apps, the selected apps have obvious candidate tasks to dispatch to other devices as we discuss for the Counting Beads example. We constructed IDTD apps from them

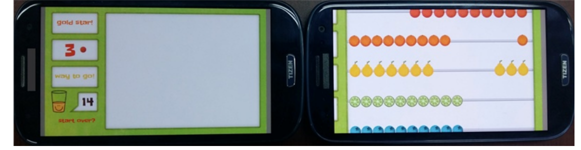

Figure 5: IDTD app constructed from Counting Beads

using the framework, and successfully performed task construction, dispatch, and communication with them.

We describe how we performed the experiments using the Counting Beads web app. We constructed an IDTD app by using the framework with Counting Beads and the following configuration:

```
cur := [Elmt Id("gpBar1")]
Elmt Id("gpBar1") :: click -> cur := [Elmt Id("gpBar1")]
Elmt Id("gpBars") :: click -> cur := [Elmt Id("gpBars")]
Elmt Id("gpRestart")::click-> cur := [Elmt Id("gpRestart")]
Elmt Id("gpTotal") :: click -> cur := [Elmt Id("quest1"),
                                 Elmt Id("quest2"), Elmt Id("quest3"),
                                 Elmt Id("gpCup"), Elmt Id("gpTotal")]
Button menu -> cur { screen: both, image: source }
```

The configuration specifies 4 static tasks as shown in Figure 4: "bar (first)" that corresponds to the first bar, which is the DOM element of id `gpBar1`, "bar (total)" containing all 5 bars, "restart button," and "scores" containing 3 score windows for the beads, the cup, and the total score window. It keeps track of a selected static task by using a variable "`cur`." When a user clicks any of the static tasks, the corresponding DOM element becomes the value of `cur`. The menu button triggers task construction by using the value of `cur` as a static task. The last line specifies that the tasks use the screens of both devices and the images of the source device.

We ran the constructed IDTD app using two Tizen phones[2] as shown in Figure 5. The phone on the left is a source and the one on the right is a target. We dispatched all 5 bars to the target, and we could play the game seamlessly using two devices. The demonstration movies of Counting Beads and another sample app are publicly available[3]. As the movies show, the performance overhead of IDTD apps is negligible.

5 RELATED WORK

Task Offloading: Several terms are used for task offloading such as computation offloading, cyber foraging, surrogate computing, and mobile cloud computing [14, 23]. Most of them have focused on augmenting mobile systems' restrained capabilities by offloading computation-intensive tasks to more powerful resources such as servers. However, offloading a task in the form of web apps that contain closures including free identifiers has not been well-explored.

Sapphire [26] supports the development of apps spanning mobile devices and clouds by proposing a general-purpose distributed programming platform with an object-based programming model. It does not support offloading a user interaction task. However, most web apps are not based on object-oriented languages, and in most cases user interaction tasks are not easily separable from other tasks. CloneCloud [7] augments the performance of mobile apps by offloading tasks on a cloned virtual machine hosted by a cloud server. However, CloneCloud requires an *offline* partitioning

[2] https://wiki.tizen.org/wiki/Reference_Device-PQ
[3] http://plrg.kaist.ac.kr/doku.php?id=home:research:idtdmovies

process, which is not possible for web apps. MAUI [9] considers energy gains in addition to performance increase to decide what task to offload. It assumes that functions to be offloaded are statically annotated, which is not valid in case of web apps. COMET [10] applies distributed shared memory (DSM) to offloading; using DSM enables COMET to provide multi-threading support with lazy release consistency, and to allow for threads to move at any point during their execution.

Application Migration: In a sense that we delegate tasks from a device to another device, our work is related to migration mechanisms [17, 19], especially web app migration [4, 16, 20]. But they consider migration of the *whole* apps only; they do not support migrating parts of the apps.

Resource Sharing: Sharing remote resources has been studied in several work [1, 2, 8]. Most of the work focus on supporting certain kinds of resources only. For example, Miracast [1] allows a mobile device to share its screen to another device, and Rio [2] supports more kinds of I/O sharing. Also, they focus on supporting I/O sharing in a *system* level, rather than in a *programming* level.

Eventual Consistency: Since the debut of the CAP (consistency, availability, and partition tolerance) theorem in 2000 [5], internet service companies become large-scale, and they often prefer availability over consistency to support their millions of users [12, 18]. In such distributed systems, eventual consistency may be more desirable than strong consistency. While most existing work on eventual consistency are ad-hoc and error-prone, Shapiro *et al.* [25] presented a principled way to guarantee eventual consistency by designing shared data types with formal conditions on them. Using various replicated data types used in practice as running examples, they showed how to formally define them and prove that they satisfy eventual consistency. Then, Burckhardt *et al.* [6] extended the work to support different objects with different replicated data types. They provided a set of axioms, from which one can define various guarantees that are stronger than the basic eventual consistency. Our IDTD framework was inspired by both work. We use the register type as it was defined by Shapiro *et al.* and we extended their or-set to support JavaScript object values. Also, our proofs are heavily based on Burckhardt *et al.*'s.

6 CONCLUSION

We proposed a novel framework to help developers to build web apps that can dispatch tasks to other devices at run time without much programming efforts. Developers can build normal web apps as usual, and the framework automatically converts them to be executable on multiple devices seamlessly. They can use the same apps on different devices without any extra work. The framework can construct tasks dynamically while preserving eventual consistency between replicated data in different devices. We evaluated the framework using 5 open-source web apps, and our preliminary experiments showed practical usability of the framework, and we made demonstration movies of the IDTD apps publicly available.

ACKNOWLEDGMENT

The research leading to these results has received funding from National Research Foundation of Korea(NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

REFERENCES

[1] Miracast. http://www.wi-fi.org/wi-fi-certified-miracast. (????).
[2] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. Rio: A System Solution for Sharing I/O Between Mobile Systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services.*
[3] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Queue* 11, 3 (2013).
[4] Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. 2011. Engineering JavaScript State Persistence of Web Applications Migrating Across Multiple Devices. In *Proceedings of the 3rd Symposium on Engineering Interactive Computing Systems.*
[5] Eric Brewer. 2012. CAP Twelve Years Later: How the "Rule" Have Changed. *IEEE Computer* 45, 2 (Feb. 2012), 23–29.
[6] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. 2013. *Understanding Eventual Consistency.* Technical Report MSR-TR-2013-39. http://research.microsoft.com/apps/pubs/default.aspx?id=189249
[7] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the 6th Conference on Computer Systems.*
[8] Jinyong Chung, Yonsuk Kim, and Donghan Kim. 2012. Portable electric device and display mirroring method thereof. (2012). US Patent App. 13/177,677.
[9] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services.*
[10] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation.*
[11] Intel Open Source.org. 2015. Web Apps 01.org. https://01.org/html5webapps/webapps. (2015).
[12] Joab Jackson. 2013. The TAO of Facebook data management. http://www.computerworld.com/article/2498193/social-media/the-tao-of-facebook-data-management.html. (2013).
[13] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *FSE'11.*
[14] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. 2013. A Survey of Computation Offloading for Mobile Systems. *Mobile Networks and Applications* 18, 1 (2013).
[15] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978).
[16] James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. 2013. Imagen: Runtime Migration of Browser Sessions for Javascript Web Applications. In *Proceedings of the 22nd International Conference on World Wide Web.*
[17] Violeta Medina and Juan Manuel García. 2014. A Survey of Migration Mechanisms of Virtual Machines. *ACM Computing Survey* 46, 3 (2014).
[18] Cade Metz. 2014. This Is What You Build to Juggle 6,000 Tweets a Second. http://www.wired.com/2014/04/twitter-manhattan/. (2014).
[19] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. 2000. Process Migration. *ACM Computing Survey* 32, 3 (2000).
[20] JinSeok Oh, Jin-woo Kwon, Hyukwoo Park, and Soo-Mook Moon. 2015. Migration of Web Applications with Seamless Execution. In *Proceedings of the 11th ACM International Conference on Virtual Execution Environments.*
[21] Changhee Park, Sooncheol Won, Joonho Jin, and Sukyoung Ryu. 2015. Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling. In *ASE'15.*
[22] Jihyeok Park and Sukyoung Ryu. 2016. Inter-Device Task Dispatch Framework for Web Applications: Supplementary. http://plrg.kaist.ac.kr/lib/exe/fetch.php?media=research:material:proof.pdf. (2016).
[23] Mahadev Satyanarayanan. 2015. A Brief History of Cloud Offload: A Personal Journey from Odyssey Through Cyber Foraging to Cloudlets. *GetMobile: Mobile Comp. and Comm.* 18, 4 (2015).
[24] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic Determinacy Analysis. In *PLDI'13.*
[25] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Technical Report RR-7506. INRIA.
[26] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. 2014. Customizable and Extensible Deployment for Mobile/Cloud Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation.*