



Research paper

COMET: Supporting seamless multi-device interaction through app component distribution

Hyeonseok Yeom^a, Hyosu Kim^b, Steven Y. Ko^c, Young-Bae Ko^{a,*}, Sangeun Oh^d^a Department of AI Convergence Network, Ajou University, Suwon-si, 16499, Republic of Korea^b School of Computer Science and Engineering, Chung-Ang University, Seoul, 06974, Republic of Korea^c School of Computing Science, Simon Fraser University, Burnaby, British Columbia, V5A 1S6, Canada^d Department of Computer Science and Engineering, Korea University, Seoul, 02841, Republic of Korea

ARTICLE INFO

Keywords:

Multi-device collaboration
Remote collaboration
Mobile computing
Pervasive computing
Ubiquitous computing

ABSTRACT

The proliferation of mobile and IoT devices has sparked growing interest in multi-device interaction, where a single app operates across multiple devices to leverage their diverse capabilities. However, the current methods of app development and usage remain bound to a single-device paradigm, rendering multi-device apps difficult to implement and deploy. This paper presents COMET, a novel mobile app framework that enables the dynamic distribution of app components across multiple devices at runtime, supporting a wide range of multi-device scenarios, including collaborative applications, smart environments, and interactions across heterogeneous personal devices. COMET enables developers to specify distributable components using lightweight annotations and employs build-time code instrumentation to automate the deployment and execution of selected components onto remote devices. This approach supports multi-device interaction with minimal developer effort and without requiring system-level modifications. To realize this, COMET addresses four key challenges: (i) the static partitioning of app components and extraction of their dependencies at build time, (ii) efficient execution of these components on remote devices, (iii) preservation of intercomponent communication across devices, and (iv) synchronization of distributed components that share a global state. We implemented a prototype of COMET on Android and evaluated it using real-world apps. Our evaluation with eight case-study apps shows that component distribution completes within 251.8 ms. A user study with 15 participants further demonstrates that COMET provides intuitive multi-device interaction with acceptable responsiveness.

1. Introduction

Over the past decade, rapid advances in the Internet of Things (IoT) and smart devices have reshaped the digital ecosystem around users. Users have grown accustomed to interacting with multiple devices simultaneously; as of 2023, each U.S. household owned an average of 17 connected devices (research, 2024) and 45% of adults used additional devices while watching TV (Perez, 2018). This trend has given rise to a new paradigm called *single-app multi-device interaction*, in which one application or app runs seamlessly across multiple devices.

This paradigm enables each app to leverage multiple devices simultaneously, thereby supporting several novel and practical use cases. One possible pattern is *resource borrowing*, where an app can leverage the resources of another device as if they were local. For example, when purchasing paid content on a smart TV through an over-the-top (OTT)

app (e.g., Amazon Prime), the OTT app's payment service can be offloaded to a smartphone, enabling the user to complete the transaction securely via a fingerprint sensor. Another pattern is *service provision*, where an app provides its functionality to another device. For instance, in a shared accommodation setting, an IoT control app can transfer its interface to a guest's smartphone, temporarily granting them control over the room's IoT devices. A final example pattern is *collaborative use*, where an app distributes interactive interfaces to multiple users, enabling them to complete a task together. For example, at a restaurant, a kiosk app can push menu interfaces to the smartphones of a group of customers, enabling them to collaboratively select items and place an order without waiting in line at the kiosk machine. These patterns demonstrate the potential of single-app multi-device interactions for enhancing convenience, efficiency, and user collaboration in everyday scenarios.

* Corresponding authors.

E-mail addresses: sooscv29@ajou.ac.kr (H. Yeom), hskimhello@cau.ac.kr (H. Kim), steveyko@sfu.ca (S.Y. Ko), youngko@ajou.ac.kr (Y.-B. Ko), sangeunoh@korea.ac.kr (S. Oh).

<https://doi.org/10.1016/j.jnca.2026.104513>

Received 21 October 2025; Received in revised form 7 April 2026; Accepted 8 May 2026

Available online 25 May 2026

1084-8045/© 2026 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

Unsurprisingly, several technologies have been proposed to support multi-device usage. One approach modifies the underlying system internals such as the kernel or mobile platform to enable seamless coordination across devices (Amiri Sani et al., 2014; Van't Hof et al., 2015; Oh et al., 2017; Chae et al., 2018; Oh et al., 2019; Alduaij et al., 2019; Lee et al., 2021; Oh et al., 2024; Shen et al., 2007; Lee et al., 2018). Although this approach enables transparent low-level support for existing legacy apps, it faces substantial deployment challenges. In practice, modifying a system's internals requires cooperation from major mobile platform vendors (e.g., Google, Apple), who are often reluctant to adopt invasive changes because of concerns over stability and backward compatibility. Consequently, system-level approaches are often impractical for widespread adoption. Another approach involves developing custom apps tailored to specific multi-device scenarios, such as video streaming between devices (e.g., Netflix) or collaborative document editing (e.g., Google Docs). While this approach avoids modifying the underlying system, it shifts much of the complexity to the application layer. In practice, developers must explicitly implement and maintain cross-device logic, including coordination, state synchronization, and error handling. This substantial engineering burden not only increases development cost, but also tends to tightly couple the implementation to specific use cases, thereby limiting the generality and reusability of such solutions.

In this paper, we present **COMET** (Component Migration and Execution with Transparency), a novel mobile app framework designed to enable seamless single-app multi-device interaction. To overcome the limitations of previous approaches, our system is guided by the following key design goals: (i) *Non-invasiveness*: The system should support multi-device interaction without modifying the underlying system, ensuring high deployability. (ii) *Ease of development*: It should significantly reduce the engineering burden on developers by enabling multi-device interaction with minimal code changes. (iii) *Versatility*: It should flexibly support a wide range of functionalities, including both system-level (e.g., sensor, camera) and app-specific (e.g., payment) functionalities, across multiple devices. (iv) *Responsiveness*: It should react promptly to user input across devices, with an average response time within 200 ms to meet user expectations (Card, 2018; Nielsen, 1994).

To satisfy these design goals, COMET enables the individual components of a mobile app to be dynamically distributed and executed across multiple devices. Modern mobile apps typically comprise multiple components,² each contributing to specific functionalities. By leveraging this modular structure, apps can distribute components to share system resources on guest devices or provide app-specific services beyond the host device. COMET augments apps with the ability to distribute components to nearby devices through code instrumentation, addressing the following four key technical challenges:

- **App component partitioning**: Extracting components from an app requires tracking all associated dependencies, including code, resource, and other dependencies. Doing so at runtime demands elevated system privileges to access internal app states and could incur significant performance overhead. By contrast, COMET adopts a static partitioning approach at build time, using a compiler module that analyzes dependencies and generates distributable bundles. In COMET, a bundle is a self-contained package file that is similar to an Android APK and contains the bytecode, resources, and metadata needed to run the extracted components on a guest device.

¹ Our framework is distinct from the COMET (Gordon et al., 2012), which addresses a different problem domain.

² In Android, each app consists of diverse components, such as Activity (UI screens), Service (background tasks), ContentProvider (data access interfaces), and BroadcastReceiver (broadcast event handlers).



Fig. 1. Example of COMET usage demonstrating multi-device interaction.

- **Remote component execution**: Partitioned components must be executable on guest devices without system-level modifications and with minimal launch overhead. To support this, COMET provides a lightweight runtime environment that efficiently supports the seamless execution of distributed components on the guest side while minimizing bundle installation and network transfer costs. As a result, COMET effectively reduces runtime overhead during component distribution, enabling components to be launched promptly and supporting responsive multi-device interaction.
- **Cross-device component interaction**: To provide the illusion that all distributed components run within the original app, COMET must correctly maintain inter-component interaction across devices. Hence, it transparently extends Android's binder channels, which were originally designed for local communication, to operate across devices, seamlessly redirecting method calls between components as remote procedure calls (RPCs).
- **Global state synchronization**: When multiple components concurrently access a shared global state, apps typically rely on synchronization constructs such as locks to ensure correctness. These semantics must be preserved even when the components are distributed across devices. To achieve this, COMET extends the existing synchronization semantics across devices, ensuring that the shared state remains consistent, even under concurrent access by distributed components.

We implemented a prototype of COMET as an application-level system on real Android devices (Google Pixel 6 and Pixel tablets), without requiring any system-level modifications. Using this prototype, we conducted a comprehensive evaluation using case study apps that reflect practical multi-device scenarios. Our experiments show that distributed components launch promptly on guest devices (within 251.8 ms), and cross-device interactions occur seamlessly with a low response latency (within 104.9 ms). In addition, a user study with 15 participants confirmed that COMET provides an intuitive and useful multi-device experience across diverse apps.

2. Use cases

To illustrate the practical potential of single-app multi-device interaction, this section details how COMET supports three types of use cases: resource borrowing, service provision, and collaborative use. Fig. 1 shows an example of multi-device interaction with COMET.³

Resource borrowing. Users often face hardware capability limitations, depending on the device used. In such cases, COMET enables apps to borrow system-level resources from nearby devices. For example, when purchasing premium content through an OTT app (e.g., Amazon

³ See <https://sites.google.com/ajou.ac.kr/comet> for our demo video.

Prime) on a smart TV, entering payment credentials via remote control is cumbersome and prone to error. Moreover, smart TVs typically lack biometric sensors, which limits their secure payment options. Using COMET, the OTT app can offload its payment component to a smartphone, enabling users to complete the transaction using fingerprint authentication. As another example, smart TV games are often constrained by limited input methods because most remote controls do not support inertial sensors or touchscreen input. COMET enables these games to use smartphones as input controllers by distributing the control components to them, thereby supporting motion-based input through built-in sensors such as accelerometers and gyroscopes.

Service provision. There are also many cases in which apps need to provide their functionality to nearby devices in a temporary or context-aware manner. COMET supports this by enabling apps to distribute only the relevant components to external devices, allowing service access beyond the host device. For instance, in a short-term letting scenario such as Airbnb, a host may wish to grant temporary control of in-room IoT devices to a guest. Existing solutions often require guests to install a specific app or use a shared tablet. However, these are impractical because users are typically reluctant to install apps for short-term use, and dedicated devices impose additional costs on the accommodation host. Using COMET, the host's controller app can transfer only its control interface to the guest's smartphone, enabling convenient IoT access without app installation or additional hardware. As another example, in large public venues such as malls or airports, users often seek indoor navigation to locate specific facilities. Rather than requiring users to install a full-featured venue app or rely on web searches, COMET enables navigation components to be temporarily delivered from venue-owned devices (e.g., kiosks) to users' phones, providing immediate access to personalized guidance.

Collaborative use. Some use cases involve multi-user collaboration; however, existing mobile apps are typically designed for single-user interactions on a single device. COMET enables collaborative use by distributing the components of a single app to multiple user devices, allowing users to interact with the app simultaneously. For example, at a restaurant, a group of customers may need to place an order together using a self-service kiosk. This often requires them to wait in line, gather around the kiosk, and interact with it individually. Such an experience is both time-consuming and inconvenient. With COMET, the kiosk app can distribute its ordering interface to each customer's smartphone, allowing them to browse the menu, select items, and place the order collaboratively, without waiting in line or crowding around the kiosk.

3. Related work

App-level multi-device support. Several custom apps (Google, 2025a,b; Netflix, 2025; Steam, 2025) support multi-device usage by leveraging cloud services for state synchronization. This enables users to switch seamlessly between devices or collaborate on shared content in real time. However, these approaches require developers to manually implement cross-device logic, resulting in a high engineering burden and limited applicability. By contrast, COMET reduces this burden by allowing developers to simply annotate distributable components. It then automatically generates the necessary distribution logic, rendering building multi-device apps considerably easier.

Multi-device system design. Many previous studies have explored system-level approaches to enable the sharing of various resources across devices. Rio (Amiri Sani et al., 2014) and M2 (AlDua'ij et al., 2019) enable remote I/O sharing by splitting the different layers of the I/O software stack, whereas Mobile Plus (Oh et al., 2017) extends Android's RPC scheme to allow cross-device access to system and app functionalities. MobiUS (Shen et al., 2007) focuses on supporting high-resolution screen sharing between devices. DynaMix (Chae et al., 2018) enables cross-device resource sharing by dynamically integrating computation and I/O resources. SAMD (Lee et al., 2018) is a system-level

framework that decomposes mobile apps into controller units, thereby enabling execution across devices. Although these systems enable rich multi-device interactions, they require modifications to the core parts of the mobile platform, which makes deployment infeasible without the support of platform vendors such as Google and Apple. In parallel, UI-centric systems such as FLUID (Oh et al., 2019), FLUID-XP (Lee et al., 2021), and A-Mash (Lee et al., 2022) enable the distribution of UI elements across devices to improve the user experience. However, these approaches also require system-level changes and allow guest devices to function only as secondary displays. In contrast, COMET supports component-level distribution without any system modification, enabling apps to utilize a wide range of hardware resources on guest devices such as sensors and cameras.

New programming models and tools. Similar to COMET, several prior studies have proposed new programming models or tools to simplify the multi-device app development. Tap (AlDua'ij and Nieh, 2021) introduces an app framework that enables cross-device functionality by virtualizing system resources, thereby allowing developers to use remote devices through platform-like APIs. However, despite its familiar interface, Tap still requires developers to manually implement cross-device logic in their code, and is limited to system-level resources. Other tools such as UIWear (Xu et al., 2017), CollaDroid (Zheng et al., 2017), and UI-Scissor (Ha et al., 2024) aim to simplify multi-device app development by distributing UI elements or generating companion interfaces (e.g., for smartwatches). However, these tools focus solely on UI distribution and are limited to display-centric use cases. In contrast, COMET automatically generates the required cross-device logic through lightweight annotations and supports both system-level resources and app-level functionalities (e.g., payments), enabling a broader spectrum of multi-device interactions.

App and thread migration. Prior studies have explored app- or thread-level migration mechanisms to enable flexible execution across devices. Flux (Van't Hof et al., 2015) supports the migration of an Android app to another device at runtime to utilize its screen. However, it is limited to a single active surface, which makes it unsuitable for concurrent use in multiple devices. Other studies (Cuervo et al., 2010; Chun et al., 2011; Gordon et al., 2012) have proposed thread migration techniques to offload computationally heavy tasks to more capable devices such as cloud servers, primarily aiming to enhance performance. However, these approaches do not consider interactive or collaborative multi-device usage, which is central to the design of COMET.

4. Background

To explore the design space of app component distribution, we targeted Android apps comprising multiple components. This section provides a brief overview of Android's component architecture, which forms the basis of the system design of COMET.

App component types. Each Android app consists of four primary component types, each serving a distinct role: Activity (user-facing screens), Service (background tasks, such as sensor data handling or payments), ContentProvider (structured data access, such as contacts or calendars), and BroadcastReceiver (system or custom broadcast handling, e.g., low-battery warnings).

Build-time component packaging. During the app build process, each component is packaged with the following elements required for execution: code logic, resources, and metadata. First, the code logic is written in Java/Kotlin source files, compiled into bytecode, and bundled into the app package along with the component's class definition and its dependencies on other classes, Android application programming interfaces, and third-party libraries. Second, resources such as user interface (UI) layouts, images, and strings are referenced through auto-generated identifiers and remain immutable at runtime because they are statically packaged during the build. Finally, each component's configuration is explicitly declared in the app's manifest. It specifies metadata such as component type, permissions, activation

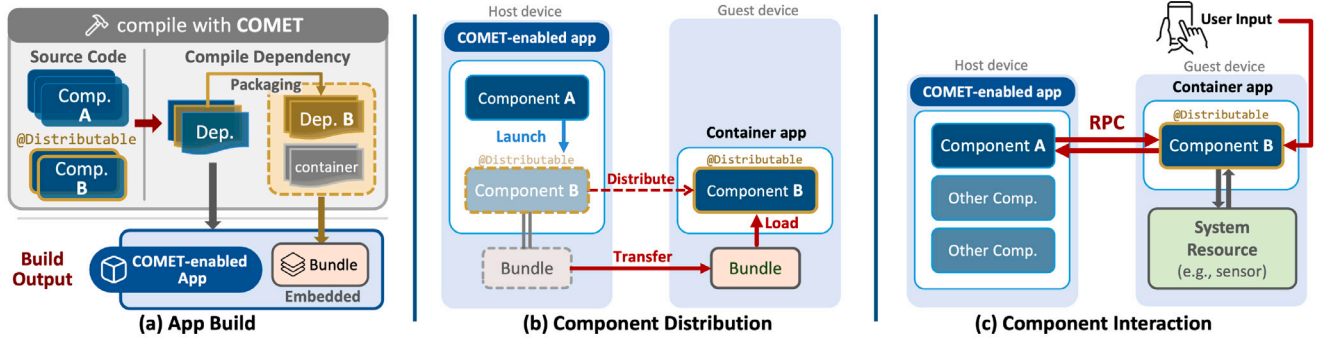


Fig. 2. COMET workflow overview (Note that the device pairing phase is omitted for concise illustration.).

rules (e.g., when it should be launched), and lifecycle attributes. These metadata ensure proper registration and execution on the Android platform.

Component registration and execution. For a component to be executed at runtime, it must first be registered with the Android platform during installation. In this phase, the package manager, an Android core system service, parses the manifest, extracts and stores component metadata, and then performs ahead-of-time (AOT) compilation and resource unpacking to ensure faster execution. At runtime, the components are activated through *intents*, asynchronous messages that specify the target to be launched. Intents may originate from other components in the same app or from different apps, enabling one component to trigger the execution of another. Once a component has been activated, the platform manages the component through its lifecycle callbacks (e.g., `onCreate()`, `onStart()`, `onDestroy()`), which handle initialization, execution, and termination.

5. System overview

To support the use cases described in Section 2, we present COMET, an app framework that augments mobile apps by offloading their functionalities to nearby devices. Without any system-level modifications, COMET enables certain functionalities of an augmented app to operate seamlessly on guest devices as if they were running within a single app boundary. To achieve this, we use Android's app components as the unit of distribution, because each functionality is fundamentally realized through one or more components. This section presents an overview of COMET.

5.1. System workflow

Fig. 2 illustrates the four-phase workflow of COMET.

App build. COMET provides a *plug-in compiler module* that integrates into the Android build toolchain, enabling developers to easily augment their apps with component-distribution capabilities. Developers simply annotate the components they wish to distribute at runtime using a directive (`@Distributable`). The compiler then packages the annotated components into a self-contained bundle along with an auxiliary code that runs on guest devices to provide a runtime environment (called a *container*) and support interaction with the host app. This bundle is then embedded into the final app package (i.e., the APK file) along with the rest of the host app. The augmented app can be published through app stores and installed on user devices.

Device pairing. Similar to any other multi-device system, COMET requires pairing between the host and guest devices. The COMET *daemon* running on the host device scans the local network and displays a list of available guest devices. Once a user selects one or more guests, the host establishes a secure network connection with each guest's daemon via an authentication process. These daemon processes act as lightweight middleware with no system-level privileges beyond basic

network access and can be installed on user devices via app stores in advance.

Component distribution. When a distributable component is triggered to launch while using a COMET-enabled app, COMET prompts the user to select an execution device, either the local device or one of the paired guest devices. If a guest device is selected, COMET transfers the bundle containing the triggered component to that device. Once the bundle has been received, the COMET daemon on the guest installs it using the Android package manager, thereby rendering it executable. The daemon then launches the container included in the bundle, which loads all the necessary code logic and resources into memory to construct the runtime environment. Finally, the container instantiates and starts the component by issuing an intent to the Android system. We refer to components that run on guest devices as *guest components*.

Component interaction. When running the guest components, COMET enables the user to interact with the host app through both host and guest components simultaneously across devices as if they were running on the same device. For example, in a sensor-based TV game, a smartphone can execute an offloaded service component that continuously processes local sensor data (e.g., gyroscopes) and sends control signals to a game-rendering component running on the TV. To support this, COMET establishes a cross-device channel that enables the host and guest components to invoke each other's methods transparently via RPCs.

5.2. System design

To realize the COMET execution model, we address the following four key challenges:

C1. How can components be partitioned from an app? To support component distribution, it is necessary to partition a subset of the components from the host app. However, this is nontrivial because most components are customized in an app-specific manner and tightly coupled with complex dependencies on code logic, resources, and metadata. Such tight coupling renders runtime partitioning impractical because it would incur significant performance overhead for dependency tracking and require system-level privileges to access app internals. To overcome this, COMET performs component partitioning ahead of time during the build phase using static analysis. Developers simply annotate the target components for distribution using a specific directive, and based on these annotations, COMET analyzes each component's dependencies and packages them into a distributable bundle. Notably, some dependencies such as the required permissions or resources for each component are not explicitly declared in the source code or manifest. To identify them, COMET further performs a complementary analysis that maps platform API calls to the required permissions and resolves resource identifiers to actual files. This ensures that the bundle includes only the necessary dependencies, minimizing bundle size and avoiding unnecessary privilege escalations on guest devices.

C2. How can the components be distributed to other devices and run with a low overhead? Once the bundle has been transferred to the guest device, COMET must provide a runtime environment to execute the distributed components without system-level modifications. A straightforward strategy would be to embed a lightweight container app within the bundle along with all distributable components and install them together as a regular app. In this setup, the container serves as the runtime environment, treating guest components as if they belong to it. Upon launch, the container initializes the runtime environment and instantiates the triggered components. However, this approach has two limitations. First, the full installation of the bundle triggers time-consuming system-level routines, such as AOT compilation and resource unpacking, resulting in considerable launch latency. Second, packaging multiple components into a single bundle can result in a large binary file, leading to substantial network transmission overhead.

To overcome these limitations, COMET adopts a lazy-loading execution model by decoupling the container and components into separate sub-bundles: (i) a container sub-bundle that provides the runtime environment, and (ii) several component sub-bundles, each containing the code and resources associated with a specific functionality. When a component is triggered, COMET transfers and installs the container sub-bundle on the guest. This sub-bundle contains only the metadata of distributable components, without their code and resources. This enables COMET to register the required components with the Android platform without incurring AOT compilation or resource-unpacking overhead. The necessary component sub-bundles are fetched in parallel while installing the container, effectively hiding the network transfer latency. Once the container has been launched, it dynamically loads the required components into memory. Consequently, this lazy-loading strategy significantly reduces both launch latency and transmission overhead while maintaining seamless execution across devices.

C3. How can seamless interaction between components be enabled across devices? To maintain the illusion that all components of the host app are running on a single device, COMET must support seamless interaction among the components even when they are distributed across different devices. This is challenging because most apps are typically developed under a single-device programming model, in which all components reside in the same process and communicate through local method calls. COMET addresses this challenge by extending Android's binder channels, which were originally designed for local inter-component communication, to operate transparently across devices. Specifically, during the build phase, COMET rewrites functions that establish binder channels by passing proxy objects (e.g., `onBind()`), enabling these proxies to be forwarded to guest components over the network. At runtime, this establishes a cross-device channel between the host and guest, enabling the host component to invoke methods on the guest component via the RPC as if they were local (and vice versa). This mechanism enables seamless inter-component interaction without requiring any changes to the app logic.

C4. How can the consistency of the shared global state be maintained across devices? Several apps rely on global states such as static variables, app preferences, or singleton objects, which may be accessed concurrently by multiple components. To ensure correctness, developers typically use synchronization constructs such as locks or synchronized blocks. However, such mechanisms are only effective within a single device and do not extend across device boundaries. When components are distributed, this can lead to race conditions and inconsistent states. Moreover, when synchronization is coordinated across devices, lock acquisition and release may involve extra network round trips. Under high concurrency, this overhead can amplify and potentially impact responsiveness. To preserve synchronization semantics across devices, COMET analyzes the host app at build time to identify synchronization-relevant codes. It then instruments the code to replace local synchronization operations with cross-device equivalents, such as

```
@Distributable(module = "payment")
class PaymentActivity: Activity() {
    // This component is part of the "payment" module.
}

@Distributable(module = "payment")
class PaymentTransService: Service() {
    // This component belongs to the same "payment" module.
}
```

Fig. 3. Annotation of app components for distribution.

transforming `lock()` into `remoteLock()`. These variants acquire the corresponding lock on the device where the global state is stored, thereby blocking other devices until the lock is released. This transformation applies to a wide range of synchronization techniques, including explicit locks, synchronized blocks, and thread-safe data structures. To avoid inflating the synchronization scope, COMET classifies synchronization-relevant code paths at build time and applies the transformation only to the identified critical sections. Consequently, COMET ensures that components can safely and transparently coordinate access to the shared state across devices.

6. Selective component partitioning

This section describes how COMET partitions app components to enable component-level distribution. To support responsiveness, COMET aims to isolate a minimal-yet-complete set of dependencies for each component. To this end, COMET performs selective partitioning at build time using three steps: (i) developer-driven component selection, (ii) static dependency analysis, and (iii) bundle packaging.

6.1. Developer-driven component selection

COMET enables developers to designate distributable components through a simple annotation-based interface. Each target component is marked with the `@Distributable` directive and grouped by functionality using the `module` attribute. For example, as shown in Fig. 3, `PaymentActivity` and `PaymentTransService` are annotated as part of the same `payment` module, indicating that they should be packaged and distributed together as a single-functionality unit. This interface enables developers to define distribution boundaries explicitly without modifying the existing app logic, while also providing COMET with the necessary context for dependency analysis and bundle generation.

6.2. Static dependency analysis

Once distributable components have been designated, COMET performs a static dependency analysis to identify all the code, resources, and metadata required for each component's correct execution. Components operate through lifecycle callbacks such as `onCreate()` and `onCreate()`; hence, COMET begins by identifying all classes and methods that are transitively reachable from these entry points. To do so, it first collects abstract syntax trees (ASTs) generated by the Android build toolchain for all source files. These ASTs capture both the syntactic and semantic structures of the app, enabling static reasoning regarding control and data flow. COMET then constructs a call graph for each component using class hierarchy analysis (CHA) (Dean et al., 1995), a conservative technique that captures all possible call targets based on the class inheritance structure. Each call graph captures all classes and methods that may be transitively accessed during the execution of the component, and serves as the basis for identifying its code-level dependencies.

However, using CHA to analyze modern mobile apps presents several challenges owing to their complexity. First, many Android apps are polyglot, written in a combination of Java, Kotlin, and C++. As CHA operates within a single language, it cannot track method calls across language boundaries. To overcome this, COMET constructs separate call graphs for each language using its respective ASTs and then merges them into a unified cross-language graph. This merging is performed by matching shared symbol signatures or applying language-specific linking rules. For instance, native C++ methods exposed via the Java Native Interface (Liang, 1999) follow a standardized naming convention that enables COMET to resolve their corresponding Java methods. This approach enables the accurate tracking of code dependencies across language boundaries.

Second, call graphs do not explicitly expose several noncode dependencies. A key example is permissions, which are required to access system-level resources (e.g., cameras, location). In Android, such permissions are declared globally in the manifest and are not tied to individual components. Consequently, determining which subset of permissions is required by a particular component is nontrivial. If COMET were to include all declared permissions to a bundle, it could lead to unnecessary privilege escalation on guest devices, violating the principle of least privilege. To avoid this, COMET maintains a mapping between the Android platform APIs and their required permissions. By traversing the call graph and inspecting all invoked platform APIs, COMET infers the minimal set of permissions required for each component. In addition to permissions, noncode dependencies on app resources (e.g., images, media files) are also invisible in the call graph because they are accessed through automatically generated identifiers (e.g., the `R` class) rather than explicit file paths. To capture these, COMET cross-references the identifiers with the build system's resource definitions to determine the files used by each component. This selective inclusion helps minimize bundle size by avoiding unnecessary resources.

6.3. Bundle packaging

Based on the results of the dependency analysis, COMET generates a self-contained bundle that encapsulates all necessary code, resources, and metadata for the designated distributable components. First, COMET consults previously generated call graphs to determine the classes that should be packaged. If a class originates from the app itself or a third-party library, its bytecode is replicated into the bundle. Conversely, classes provided by the Android platform are excluded because they are assumed to be available on guest devices via the guest-side Android runtime. Notably, when app-defined classes are included, COMET rewrites some portions of their code to enable RPC-based interaction and preserve synchronization semantics across devices. These transformations are described in Sections 8 and 9.

For noncode dependencies, COMET includes only the minimal set of app resources and metadata required for execution. Resource files are selectively packaged based on dependency analysis to reduce bundle size. For the metadata, COMET generates a new manifest file that declares the specific permissions, activation rules, and configuration values required for each component and embeds it within the bundle.

Finally, the bundle includes the auxiliary runtime code required to operate a lightweight container on the guest device. This container initializes the runtime environment and manages the lifecycle of the distributed components. To further optimize runtime performance, COMET organizes the bundle into multiple sub-bundles, separating the container logic from the component-specific code and resources. This bundling strategy is described in detail in Section 7.

7. On-demand component distribution

When component distribution is requested during the execution of a COMET-enabled app, COMET must promptly deliver the corresponding bundle to a guest device and launch it with minimal delay. To

enable this without any system-level modifications, COMET provides a lightweight runtime environment that efficiently supports the seamless execution of the distributed components on the guest device. This section describes how COMET triggers component distribution and organizes the runtime environment to support remote execution.

7.1. Within-app intent redirection

To initiate component distribution at runtime, COMET intercepts component launch requests issued by the host app. In Android, components are typically launched by passing intent messages to the system via specific platform APIs such as `startActivity()`, `startService()`, or `bindService()`. To enable interception without modifying the Android platform, COMET rewrites these launch calls at build time to invoke functions provided by COMET, wrapper functions (e.g., `COMET.startActivity()`). These wrapper functions act as decision points for intent redirection. When invoked, they inspect the intent to determine whether a target component is marked as distributable. If so, COMET prompts the user to select a device on which to execute the component; otherwise, the intent is forwarded unchanged to the system for local execution. Once a guest device has been selected, the intent is transmitted accordingly. Importantly, because redirection is performed at the module granularity, all subsequent components belonging to the same module are automatically distributed to the same device.

7.2. On-demand runtime environment

As discussed in Section 5.2, a naive way to provide a guest-side runtime environment is to embed a container app within the bundle along with all distributable components and install them as a regular app. However, this approach incurs substantial latency due to installation overheads such as AOT compilation and resource unpacking. In addition, packaging multiple components into a single bundle can lead to large binaries and increase transmission delays.

To overcome these limitations, COMET adopts an on-demand runtime model based on lazy loading, which decouples the lightweight container logic from component-specific parts. Instead of transferring and installing the entire bundle simultaneously, COMET organizes it into multiple sub-bundles and dynamically loads only the necessary parts at runtime. These sub-bundles consist of two types, a *container sub-bundle* and multiple *component sub-bundles*. The container sub-bundle encapsulates the minimal runtime logic required to launch the distributed components. By contrast, each component sub-bundle contains the code, resources, and metadata needed for components associated with a specific functionality. This modular organization enables COMET to handle the container and components independently, facilitating fine-grained control over what is loaded and when. Consequently, this design reduces both the installation overhead and transmission latency during component distribution, as detailed in the descriptions of the following two techniques.

Code-free component registration. As shown in Fig. 4, when an intent is redirected to a guest device (step 1), COMET first transfers the container sub-bundle to the guest (step 2) and installs it. As this sub-bundle contains only minimal runtime logic and excludes the code and resources of the guest components, it can be installed quickly with minimal overhead, typically within 281 ms on our test device (Google Pixel 6). However, to be launchable, all guest components must be registered with the Android platform as if they belong to the container. Hence, COMET leverages a special installation routine used in Android's dynamic plug-in features (i.e., play feature delivery (Google, 2025c)). This mechanism enables components to be registered with the system at install time without including their actual code or resources in the app package. To utilize this, COMET embeds only the metadata of all guest components, such as component properties and required permissions, into the container's manifest at build time. This manifest

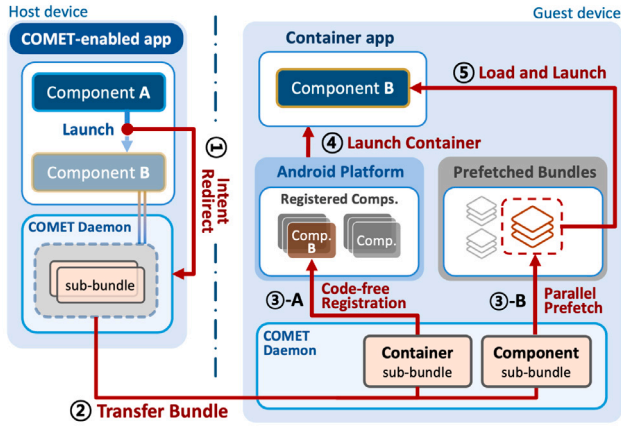


Fig. 4. On-demand runtime environment. Steps 3-A and 3-B are executed in parallel.

enables the Android platform to recognize and register guest components as if they were part of the container (step 3-A), even though their implementations have not yet been delivered. As a result, COMET can bypass time-consuming steps such as AOT compilation and resource unpacking.

Note that skipping these steps does not affect the correctness of component execution. This is because resource unpacking is simply deferred until the component is loaded at runtime.

Parallel component prefetching. While the container sub-bundle is being installed, COMET transfers the requested component sub-bundle to the guest device (steps 2 and 3-B). This parallelism effectively conceals the network latency behind the container installation time. Once the container has been installed, it is immediately launched by the COMET daemon (step 4). The container then dynamically loads all components in the prefetched sub-bundle into memory using a custom class loader and unpacks their resources to make them accessible (step 5). As all guest components have already been registered with the platform via code-free registration, the container can issue a redirected intent to the guest-side Android platform as if the target components were locally installed. The platform then launches the target component normally, completing the remote execution process with minimal startup latency.

To further reduce the latency for future launches, the container prefetches additional component sub-bundles in the background, even if they are not immediately requested. This proactive strategy leverages the idle time during the execution of the initial component to prepare subsequent components in advance, thereby improving responsiveness in future component distributions. Moreover, as long as the host app is not updated, both the installed container and prefetched components remain on the guest device and can be reused in subsequent app usage sessions (called a *warm start*). During warm start, COMET first validates cached sub-bundles against the host app's version metadata (e.g., `versionCode`). If the host app has been updated since the cache was created, COMET invalidates the cache and re-distributes the required sub-bundles to avoid stale code and resources. Otherwise, COMET reuses the cached sub-bundles, enabling fast and seamless remote launches without redundant installations or data transfers.

8. Transparent cross-device interaction

When components are split among the host and guest devices, the function calls between them must behave as if they were local. Ensuring such transparency is essential for maintaining correct app behavior without requiring any changes to the existing app logic. To achieve this, COMET automatically redirects local method calls to cross-device RPCs when necessary. This section describes how COMET supports function call redirection and RPC execution in a transparent manner.

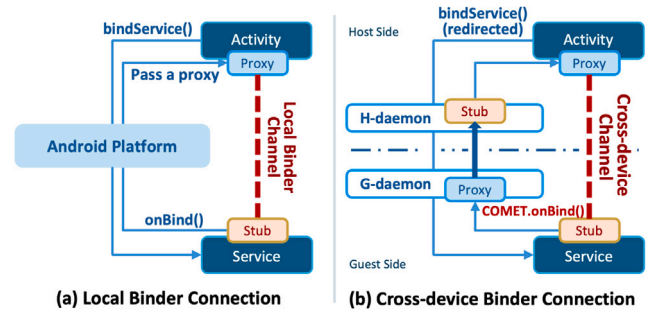


Fig. 5. Local and cross-device binder connections.

8.1. Function call redirection

To support cross-device function calls, one naive approach would be to instrument all inter-component calls at build time to perform RPCs. However, this approach has two major limitations. First, extensive modification of the app code may be required, which is both error-prone and difficult to maintain. Second, it introduces runtime overhead, as each instrumented call should dynamically determine whether the callee resides on a remote device and decide whether to transmit an RPC.

COMET instead adopts a more efficient approach by leveraging Android's inter-component communication model, which is known as the *binder*. As illustrated in Fig. 5(a), binder communication relies on a proxy-stub pair: a proxy object is passed between the components to establish a binder channel, enabling the receiver to invoke functions on the sender, even across process boundaries. COMET extends this model to support cross-device communication by modifying only the functions responsible for passing proxy objects (e.g., `onBind()`). At build time, COMET rewrites these functions to forward proxy objects to the guest device through the COMET runtime. This transformation establishes cross-device communication channels between distributed components, enabling their function calls to be transparently routed over the network without modifying individual call sites.

For example, Fig. 5(b) illustrates the manner in which an activity on the host device binds to a service on a guest device. When the activity calls `bindService()`, COMET triggers the remote execution of the service on the guest, as described in Section 7. Normally, the service's `onBind()` method would be invoked to return a proxy object for establishing a communication channel. However, because COMET rewrites this call at build time, the activity instead invokes `COMET.onBind()`, which routes the proxy through the COMET runtime. This instrumented function passes the proxy to the guest-side daemon (G-daemon), which locally binds to the service, and forwards the proxy over the network to the host-side daemon (H-daemon). As this proxy is device-specific and thus invalid on the host, the H-daemon generates a new stub-proxy pair that is valid on the host and returns the new proxy to the activity. This establishes a local binder channel between the activity and H-daemon, whereas the G-daemon maintains a connection to the guest-side service. Finally, the two daemons bridge these local channels over the network, enabling the seamless remote invocation of service methods from the activity.

8.2. RPC execution support

When a function call is redirected across devices, COMET serializes the function name and its arguments into an RPC message and transmits it to the guest device. Upon receipt, COMET must ensure that the corresponding function is executed correctly on the guest. We broadly classify the correctness challenges into two categories. The following

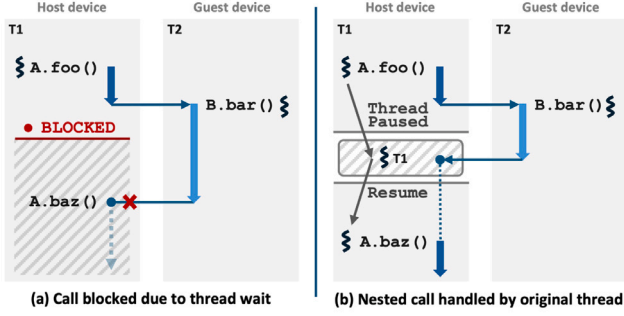


Fig. 6. Preserving a function call sequence across devices.

discussion assumes that the host initiates an RPC, but the same principles apply in the reverse direction. The first challenge arises when the target function includes reference-type arguments such as uniform resource identifiers that are valid only on the host device. If such references are naively passed to the guest, RPC execution may crash. To address this issue, COMET employs a reference resolution mechanism that transfers the referenced resource object to the guest and generates a new reference that is valid in the guest context. The original argument in the RPC is then replaced with the new reference, ensuring that the function on the guest can access the resources correctly.

Second, preserving the correct execution context is challenging when function calls form a nested sequence across devices. For example, as shown in Fig. 6, we consider a nested call sequence between components A and B on different devices: `A.foo()` invokes `B.bar()`, which in turn calls back `A.baz()`. On a single device, such nested calls are executed using the same thread via a unified call stack. However, in a distributed setting, the stack is fragmented across devices, thereby breaking the continuity of the original execution flow. In this example, the host thread T_1 waits for the return value after invoking `B.bar()`, whereas the guest thread T_2 executes `B.bar()`. When T_2 invokes `A.baz()` via RPC, it must be handled by T_1 , particularly if T_1 is the UI thread. Otherwise, running it on another thread could violate Android's threading model, which requires UI operations to run on the UI thread; this could potentially cause runtime errors. However, because T_1 is blocked while waiting for the result of `B.bar()`, it cannot immediately process the nested call. To solve this problem, COMET maintains a logical mapping between threads involved in the same cross-device call chain. In this case, T_1 and T_2 are linked as part of a single logical context. When `A.baz()` is invoked by the guest, COMET temporarily wakes T_1 , which was blocked, waiting for the result of `A.foo()`, to execute the nested call. This is possible because COMET manages the blocking behavior by placing T_1 in a controlled waiting queue that allows temporary re-entry for nested invocation. Once `A.baz()` is completed, T_1 resumes waiting for the return value from `A.foo()`.

9. Global app state synchronization

App components may need to concurrently access a shared global state such as static variables or app preferences. While developers typically rely on synchronization primitives, such as locks or synchronized blocks, to ensure correctness, these mechanisms assume a single-device context. When components are distributed across devices, this assumption no longer holds, leading to potential race conditions and inconsistencies. To address this issue, COMET extends the app's existing synchronization logic across devices, ensuring data consistency even during remote access. Notably, COMET does not attempt to make unsynchronized data safe but instead preserves the synchronization semantics already defined by the developer.

To this end, COMET identifies global states at build time using points-to analysis (Stensgaard, 1996). Rather than analyzing all global

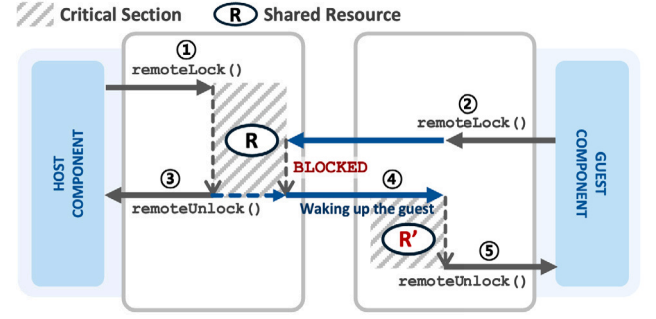


Fig. 7. Preserving synchronization across devices.

states, the analysis narrows its scope to those accessed by distributable components that may be executed on different devices. Specifically, it detects accesses to static variables, predefined global structures (e.g., singletons), and app preferences. COMET then classifies the identified states as either *encapsulated* or *non-encapsulated* based on how their data are exposed and synchronized, and it applies tailored synchronization mechanisms to each category through code instrumentation.

In the encapsulated case, the internal data of the shared global states are not directly exposed, which adheres to object-oriented design principles. Access to the data occurs only through well-defined methods (e.g., getters, setters), where synchronization logic is implemented internally to ensure consistency. To preserve these semantics across devices, COMET retains actual object instances only on the host and creates corresponding *virtual objects* on the guest. These virtual objects are placeholders that expose the same interface and forward all interface method calls to the host via RPC. This design enables guest components to transparently access global states while ensuring that synchronization remains governed by the host-side logic. To enable this, COMET selectively replaces references to encapsulated global objects on the guest with their virtual counterparts, as identified by static analysis.

In contrast, non-encapsulated states expose their internal data directly to external components, often through public fields, enabling access without method calls. This implies that synchronization must be applied explicitly at each access site to prevent race conditions. To preserve these synchronization semantics across devices, COMET replicates such global states from the host to the guest and transforms local synchronization constructs (e.g., locks, synchronized blocks) into their cross-device equivalents via code instrumentation. This ensures that the updates made on either the host or guest are properly coordinated, maintaining a consistent view of shared states across devices.

Fig. 7 illustrates how two components on different devices coordinate access to a shared global state using cross-device locks. Each device maintains its own instance of the lock, but the two are logically linked to enforce mutual exclusion. When a guest-side thread invokes `remoteLock()`, COMET forwards the request to the host, which grants the lock only if the host-side instance is currently available; otherwise, the guest thread is blocked until the host releases the lock. In this case, once the host-side thread has finished its critical section, it invokes `remoteUnlock()`, releasing the lock and resuming the blocked guest thread. If the host has modified the global state during its critical section, the updated state is propagated to the guest. The same logic applies in the reverse case, in which the guest acquires the lock before the host. Note that COMET handles synchronized blocks in the same manner by inserting `remoteLock()` and `remoteUnlock()` at the entry and exit points of the block during the build phase. This enables existing synchronization constructs to function correctly across devices without the need to modify app logic.

Table 1
COMET-enabled app list for the case studies.

Type	App name	Use case scenario	Selected component	LoC	APK size (kB)	Arg. size (bytes)
Resource borrowing	WatchVOD ^a	Performing OTT Payments via fingerprint authentication on a different device	Activity for fingerprint authentication Service for payment transaction	2/1407 (0.14%)	1206/13,443 (7.6%)	198
	Pixelwheels	Using another devices sensors as a controller	Service to access tilt sensor data	1/35,628 (<0.01%)	472/36,703 (1.3%)	177
Service provision	AirStay ^a	Temporarily providing digital services (e.g., IoT, digital keys) to a guest	Activity to control IoT devices Service for notifications Service for digital key	3/9277 (0.03%)	3400/113,458 (3.0%)	313
	Calendar	Continuing to use the calendar on another device	Activity to display calendar Broadcast receiver for reminders	2/40,019 (<0.01%)	5314/33,976 (15.5%)	908
	Messenger	Continuing conversations on another device	Activity for chat room display Broadcast receiver for new messages	2/20,567 (<0.01%)	1773/31,253 (5.7%)	1686
Collaborative Use/ Content Sharing	OrderKing ^a	Allowing others to use order-related functions	Activity for placing an order Service to show order progress Broadcast receiver for order notification	3/1584 (0.19%)	1552/14811 (10.2%)	246
	Gallery	Personal photo sharing with others	Activity to preview images Content provider to share images	2/43,826 (<0.01%)	4,344/34,628 (12.6%)	2332
	FileManager	File sharing with others	Activity to show file transfer progress Content provider to share selected files	2/11,909 (0.02%)	2,434/14,938 (16.4%)	1577

^a Apps are custom-made apps; the others are open-source apps. Arg. Size: the maximum amount of data transferred for RPC arguments.

10. Implementation

We implemented a COMET prototype that enables component distribution without modifying the underlying Android platform. COMET comprises three main parts:

Compiler module: The module, implemented as a Gradle plugin (Grandle, 2025) using the Transform APIs (Kotlin, 2025a,c,b), intercepts ASTs generated by the Android compiler front-end and performs static analysis to partition components annotated with `@Distributable` into sub-bundles along functional boundaries (Section 6). For each sub-bundle, it also injects code logic to support cross-device RPCs and distributed synchronization (Sections 8 and 9). Instrumentation is done by rewriting the ASTs, which are then passed to the compiler back-end for code generation. The resulting sub-bundles are embedded into the host app's final APK.

Container: The container is a standalone Android app that loads prefetched sub-bundles using `DexClassLoader` (Section 7) and launches them via standard intents; it is also packaged into the host APK.

Daemon program: The COMET daemon runs as a persistent background service to manage runtime operations such as device pairing, container installation, component registration, prefetching, and Binder channel setup (Sections 7, 8). Container installation relies on the public `PackageInstaller` API, which prompts the user with a standard approval dialog; once accepted, the container is installed, and guest components are registered without the need for root privileges or system modifications.

11. Evaluation

Through the COMET prototype, we demonstrate its support for cross-device component distribution in practical settings. To evaluate its effectiveness, we developed eight diverse multi-device apps and used them as case studies in our experiments. All experiments were conducted on a Google Pixel 6 smartphone and Google Pixel Tablet, both running Android 14. The devices were connected to the same Wi-Fi access point with a measured throughput of 433 Mbps and a round-trip time (RTT) of 24.8 ms (average) and 23.2 ms (standard deviation). This Wi-Fi configuration serves as the default environment for all experiments unless otherwise specified.

To further validate COMET under real-world network conditions, we conducted a subset of experiments in a crowded public space (a Starbucks cafe) using cellular networks (LTE and 5G). Since cellular

environments are subject to heavy contention from nearby users, they exhibit significant fluctuations in bandwidth and RTT. In these cellular experiments, LTE achieved an average throughput of 83 Mbps (49 ms RTT), while 5G reached 370 Mbps (30 ms RTT). At the 10th–90th percentile, LTE throughput ranged from 31 to 111 Mbps, with RTTs between 39 and 51 ms; 5G throughput ranged from 313 to 530 Mbps, with RTTs between 28 and 35 ms. Using these setups, we evaluate both performance and usability to assess how well COMET supports multi-device scenarios.

11.1. Case studies

To demonstrate that COMET simplifies the development of multi-device apps, we built eight case-study apps using COMET, as summarized in Table 1. These include five modified open-source apps and three new apps developed from scratch. In these apps, up to three components could be distributed for remote execution. We confirmed that COMET successfully enabled cross-device distribution in all cases.

The LoC column in Table 1 lists the proportion of code added to enable component distribution relative to the total lines of code for each app. This typically results in only one additional line per component, demonstrating that COMET introduces minimal developer burden. The APK size column reports an increase in APK size because of the inclusion of all the sub-bundles generated by the COMET compiler. We observe that activity components tend to contribute more significantly to APK size increase because replicating them requires bundling additional UI resources (e.g., layout files, drawables). By contrast, non-activity components generally incur a smaller overhead.

11.2. Performance test

We quantitatively evaluated the ability of COMET to support multi-device operations. Each experiment was repeated 10 times across three device configurations: phone-to-phone, phone-to-tablet, and tablet-to-phone. As the results showed similar trends across configurations, we report only the phone-to-phone results for brevity.

Distribution time. To evaluate how efficiently COMET distributes components across devices, we define the *component distribution time* as the duration from when the user input triggers the component distribution to when the corresponding component is launched on the guest device. We compared the following four approaches to isolate the effectiveness of the core techniques described in Section 7:

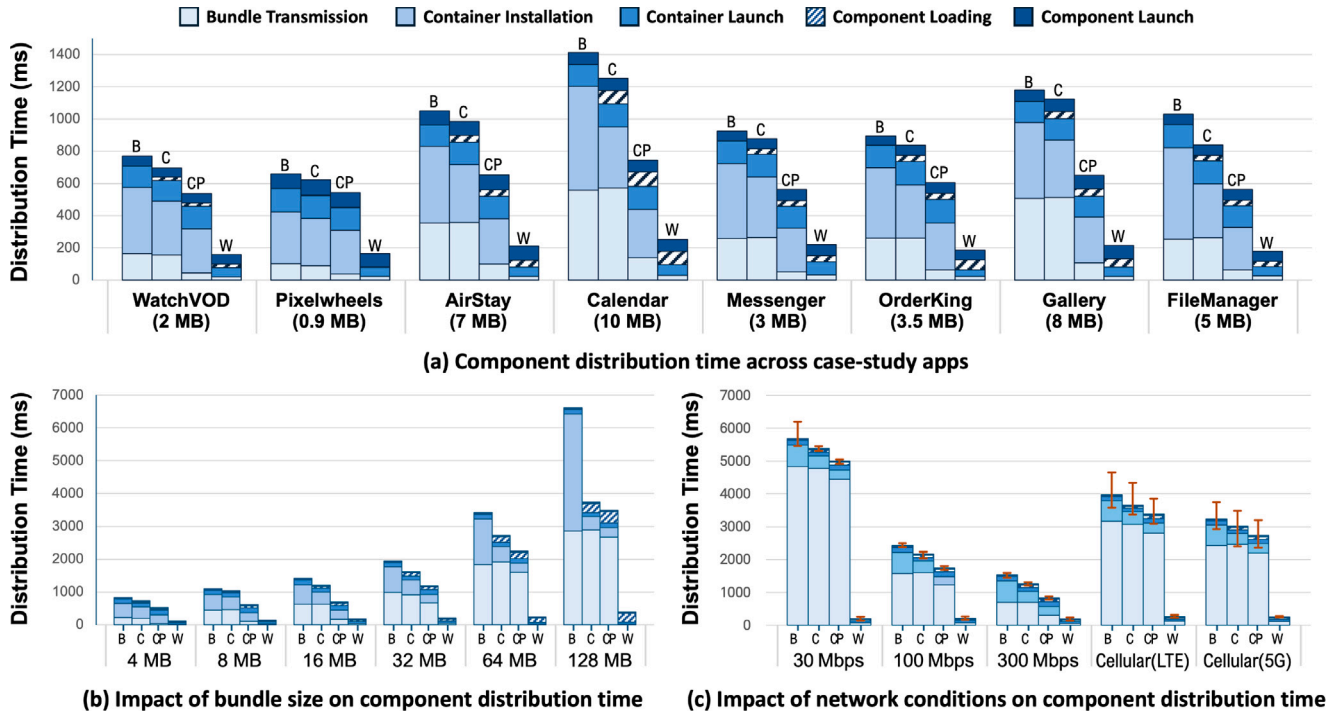


Fig. 8. Component distribution time for (a) each case-study app, where the number below each app indicates the size (in MB) of the distributed bundle, (b) varying bundle sizes, and (c) different network conditions. (BASE: B, COMET-C: C, COMET-CP: CP, and COMET-W: W.)

- BASE: All distributable components are packaged into a single monolithic bundle. During distribution, the bundle is transmitted to the guest, fully installed, and the container is launched to start the requested component.
- COMET-C: Code-free registration is used. Once the container has been installed, the component sub-bundle is transferred.
- COMET-CP: COMET-C is extended by prefetching the component sub-bundle in parallel during container installation.
- COMET-W: Sub-bundles cached on the guest device from a previous app usage session are reused (i.e., a warm start).

Fig. 8(a) shows the component distribution times for our case-study apps, divided into five stages: bundle transmission, container installation, container launch, component loading, and component launch. This breakdown reveals how each stage contributes to the total latency of different approaches. In COMET-CP, bundle transmission overlaps with container installation and launch via prefetching, whereas the other approaches execute all stages sequentially. In COMET-W, the bundle transmission stage represents only the network delay in delivering the intercepted intent from host to guest.

In BASE, packaging all the components into a single, large bundle leads to high transmission and installation costs. The installation step dominates the overall delay because of costly system-level operations such as AOT compilation and resource unpacking. Consequently, BASE exhibits the highest end-to-end distribution time of 1413.4 ms. By contrast, COMET-C reduces the distribution time by up to 18.5% by avoiding a full installation. Although COMET-C incurs a small overhead when dynamically loading the sub-bundle after the launch, this cost is far lower than the installation delay of BASE. COMET-CP improves performance by prefetching the component sub-bundle in parallel with container installation, hiding most transmission delay and reducing the total distribution time of BASE by 47.2%. Finally, COMET-W achieves the best performance, completing the distribution in as little as 251.8 ms regardless of app size or complexity, because the required sub-bundles are cached on the guest.

Although these results reveal how COMET performs across case-study apps, they do not capture how distribution time scales with

bundle size. To examine this, we created micro-benchmarks using synthetic apps with varying bundle sizes, adjusted by changing the amounts of code and resources. Fig. 8(b) plots the distribution time against bundle size for each approach. In BASE, the time increases linearly because of the increased transmission and installation overheads, whereas COMET-C maintains a nearly constant installation time, reducing the delay by up to 43.5% at 128 MB. COMET-CP further improves performance for small to medium bundles (4–16 MB) by hiding transmission with prefetching, but for larger bundles transmission dominates, yielding little gain over COMET-C. Finally, COMET-W consistently achieves the lowest latency, completing distribution in under 382.6 ms even at 128 MB.

Fig. 8(c) evaluates COMET's distribution performance when transferring a 16-MB bundle under diverse network conditions. We consider two settings: (i) bandwidth-limited Wi-Fi, where bandwidth is capped at 30/100/300 Mbps using the Linux traffic control (TC), and (ii) real-world cellular links (LTE/5G) characterized by significant bandwidth fluctuations. In bandwidth-constrained Wi-Fi settings, lower bandwidth makes bundle transmission the dominant bottleneck, increasing the end-to-end distribution latency. Nevertheless, COMET-W remains effective because it can reuse sub-bundles cached on the guest device, reducing repeated transfer overhead. For cellular networks, distribution is faster than in the most bandwidth-constrained Wi-Fi setting, but tends to be slower than stable 100 Mbps Wi-Fi due to bandwidth variability. Even so, COMET-W continues to deliver robust performance under such fluctuations. Overall, these results indicate that even in bandwidth-limited or highly variable networks, once the one-shot sub-bundle transmission cost is paid, COMET can subsequently achieve faster component distribution by leveraging cached components.

Response time. To evaluate the responsiveness of the components distributed to guest devices, we measured the *component response time*, defined as the delay from the user input on the guest device until the result appears on either device. Fig. 9 shows the average response times for various tasks across our case-study apps, broken down into network transfer, binder communication, and app-level computation, with the number of cross-device RPCs labeled above each bar and argument sizes reported in Table 1 under the *Arg. Size* column.

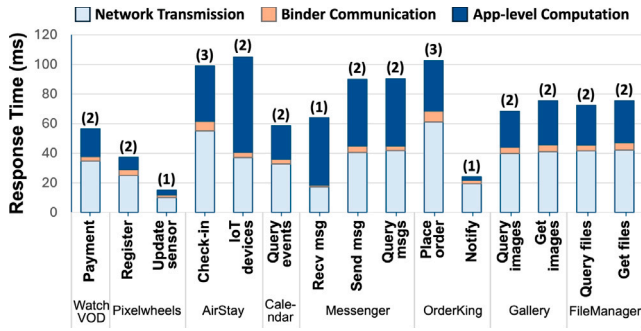


Fig. 9. Component response time across different tasks. The numbers in parentheses indicate the number of cross-device RPCs for each task.

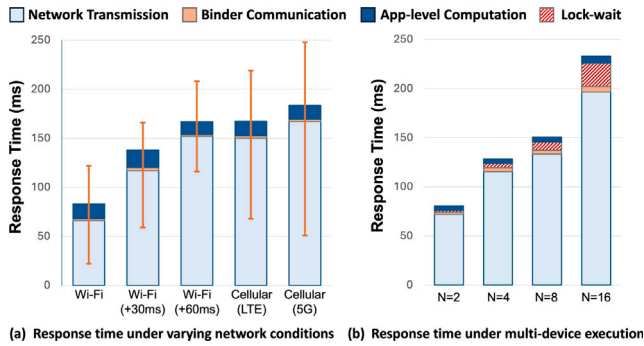


Fig. 10. Component response time under varying execution conditions. (a) Response time under different network conditions. (b) Response time under concurrent multi-device execution, where N denotes the number of guest devices.

All response times were below 104.9 ms, which demonstrates that COMET provides sufficient responsiveness for interactive use (Card, 2018; Nielsen, 1994). The response time tends to increase linearly with the number of cross-device RPCs because network latency accumulates across calls. For example, the order placement task in OrderKing and the hotel check-in task in AirStay introduce additional RPCs for progress updates or confirmations, leading to higher latency. Although such overhead is inherent to cross-device communication, it can be mitigated using advanced wireless technologies, such as Wi-Fi 6E (Mozaffariahrar et al., 2022). The IoT control task in AirStay also has relatively slow response time, mainly because of app-level delays in interacting with the external hardware.

Fig. 10(a) presents the average response time of the OrderKing ordering task under diverse network conditions. We consider two settings: (i) a controlled Wi-Fi environment, where we inject a fixed delay of 0/30/60 ms using the TC tool, and (ii) LTE and 5G networks, in which delay fluctuates significantly.

Overall, total response time shows a clear upward trend as network delay increases. Notably, although LTE and 5G exhibit relatively low link latency (with average RTTs of 49 ms and 30 ms, respectively), their observed response times are comparable to those observed in the Wi-Fi scenario with an injected delay of 60 ms. This discrepancy stems from the significant jitter (delay fluctuation) inherent in cellular networks, which degrades end-to-end interaction latency. Nevertheless, COMET remains responsive under such variability, achieving average response times of 167.4 ms on LTE and 183.6 ms on 5G.

Scalability test. We evaluated the scalability of COMET as the number of guest devices increases, using the order placement task in OrderKing. In this setup, multiple guest devices concurrently send orders to the host device. Upon receiving each request, the host updates the order state, which is maintained as a global variable (i.e., shared data).

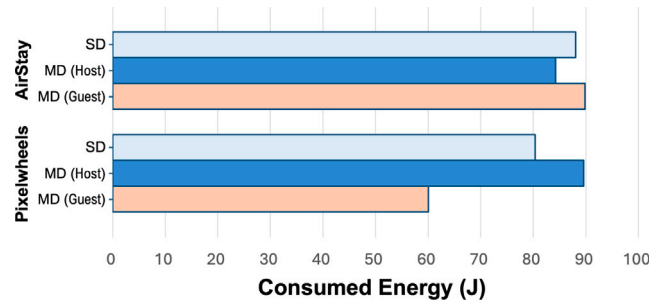


Fig. 11. Energy consumption under single-device (SD) and multi-device (MD) execution for representative distributed component types.

Fig. 10(b) shows the long-tail response time, defined as the maximum latency observed among all participating guests. As the number of guests increases, the response time exhibits an upward trend, rising from 78.4 ms ($N = 2$) to 227.2 ms ($N = 16$). Our analysis identifies *network transmission* as the primary bottleneck. When many devices send request packets simultaneously, packets accumulate in the host device's network queue, and the host processes them sequentially, delaying request delivery to the host app. The secondary factor is the *lock-wait* time. In our setting, its impact remains relatively minor because the significant network queuing delay naturally staggers request processing, often allowing previous critical section updates to complete before a subsequent request attempts to acquire the lock. However, if the app logic were to require a larger critical section, lock contention could become the dominant factor in response time. Therefore, keeping the host-side critical section short is important for scalability.

11.3. Energy consumption

We also evaluated the energy consumption of COMET during multi-device interactions. Energy usage was estimated by reading current and voltage values from `/sys/class/power_supply/battery/`.⁴ This software-based approach is widely adopted for smartphones with non-removable batteries, yielding an estimation error typically within 10%.⁵ For fair comparison, we set the display brightness of all devices to 100% and started each experiment with the battery fully charged.

Fig. 11 compares the energy consumption when running each app for 25 s under a single-device setting (SD) and a multi-device setting (MD). Since COMET's energy impact depends on the characteristics of the distributed component, we consider two representative component categories: (i) *Activity*, a UI-centric component that requires periodic on-screen updates (AirStay), and (ii) *Service*, a non-UI component that performs data processing in the background without display rendering (Pixelwheels). In the case of AirStay, the guest device exhibits energy consumption comparable to the SD baseline as the distributed Activity continues to refresh the screen. Conversely, the host device consumes 4.3% less energy than SD, being relieved of the UI update overhead. On the other hand, for Pixelwheels, the offloaded Service on the guest eliminates display refreshes, resulting in 25.2% lower energy consumption than SD. However, the host exhibits an 11.5% increase compared to SD; this is because it must maintain UI updates while additionally processing sensor data streamed from the guest.

⁴ The Power Management Integrated Circuit (PMIC) periodically reports power-related measurements (e.g., voltage, current, battery capacity) with an error rate of $\pm 1\%$.

⁵ Prior work (Nguyen et al., 2016) reports that PMIC-based energy estimation at 2 Hz yields less than 10% error.

Table 2
User study results for case-study apps.

Application	Performance		Intuitiveness [0,5]	Usefulness [0,5]	Satisfaction [0,5]
	Distribution [0,5]	Response [0,5]			
WatchVOD	4.87	5.00	4.53	4.40	4.47
OrderKing	5.00	5.00	4.53	4.07	4.47
AirStay	4.93	4.93	4.80	4.13	4.53
Pixelwheels	–	4.47	4.67	4.47	4.33

11.4. Usability study

To assess the practical benefits of COMET from an end-user perspective, we conducted a usability study involving four case-study apps selected from Table 1. The goal of our evaluation was to evaluate users' perceived performance, intuitiveness, usefulness, and overall satisfaction when interacting with multi-device apps built using COMET. We designed COMET's distribution UI as a familiar device-chooser prompt (e.g., mobile OS built-in chooser dialogs and Apple's AirDrop-style device selection) to support intuitive triggering of component distribution.

Participants and procedure. We recruited 15 participants (13 male, 2 female, mean age 27.5 years) through an online community posting. Each participant was asked to complete predefined usage scenarios for the four apps selected from the case studies. The participants interacted with each app using two connected devices powered by COMET, following a predefined task script designed to simulate realistic use cases. After completing each scenario, participants rated their experiences by answering questions about four aspects of their experience:

- *Performance of the component distribution/response:* "How much delay did you feel when distributing components to the remote device and interacting with them afterward?"
- *Intuitiveness of distribution interface:* "How intuitive do you think the interface for triggering component distribution is?"
- *Usefulness:* "How useful do you think the multi-device interaction is, compared to using a single device?"
- *Satisfaction:* "What is your overall satisfaction with using a COMET-enabled app?"

Each question was rated on a six-point Likert scale (0–5) that deliberately excluded a neutral midpoint to encourage participants to express a clear stance. Following these tasks, interviews were conducted to gather qualitative feedback regarding their experiences. Each session lasted approximately 40 min, and the participants were paid approximately 4 USD.

Results and findings. Although scores varied slightly across apps, participants generally found COMET to provide a useful and satisfying multi-device experience with acceptable performance. As shown in Table 2, all four apps received positive ratings. Specifically, the average scores for perceived performance, intuitiveness, usefulness, and satisfaction were all above 4.0 on the 6-point scale, with minimum values of 4.47, 4.53, 4.07, and 4.47, respectively (0 = *very bad*, 5 = *very good*). Note that we did not collect the perceived distribution performance scores for Pixelwheels because it offloads a background service that users would not directly perceive during interaction.

One notable finding is that Pixelwheels received the highest usefulness score despite having the lowest perceived response performance. The participants reported noticeable delays because the control signals derived from real-time sensor inputs were transmitted over the network. This latency was particularly disruptive in a fast-paced racing game and negatively affected perceived performance. Nevertheless, they appreciated the overall concept of using a smartphone as a steering controller, which resulted in the highest usefulness rating. This indicates that even with low performance, users can still find COMET-enabled apps useful.

Another interesting observation was made for OrderKing. Despite receiving the lowest average usefulness score among the four apps, it

was selected by the largest number of participants (9 out of 15) as the most useful scenario during the post-study interview. This discrepancy stems from a few particularly low usefulness ratings. In follow-up interviews, these participants explained that many restaurants already provide table-side ordering options, such as built-in tablets or QR codes, which reduce the need for OrderKing. However, they also acknowledged the limitations of these alternatives, noting that installing physical tablets at every table is expensive and QR codes are vulnerable to phishing attacks. From this perspective, they agreed that COMET-based solutions, such as OrderKing could offer a more cost-effective and secure alternative.

Below, we share some quotes from the participants for each app.

- *WatchVOD was very convenient but potentially abusable.* P4 responded that "It's great that I can securely complete payment on my phone when using a smart TV in public". Similarly, P15 noted that "Typing on a TV remote is frustrating, so being able to confirm payments on my phone really improves the experience". However, a few raised concerns about possible misuse; for instance, triggering a payment screen on another user's device to manipulate them into paying unintentionally.
- *OrderKing was the most useful scenario chosen by the majority of participants.* P2 appreciated that "Unlike current services that require installing a dedicated app, such as Starbucks, this app was very convenient because it didn't require any installation", and P8 highlighted that "It was useful to see how multiple people can collaboratively place an order from their own devices, without having to wait in line at a kiosk".
- *AirStay garnered mixed opinions on practical utility.* P6 said, "Being able to control IoT devices in a hotel room with my own phone, without installing any apps, seems like a great feature and could work well with services like Airbnb". On the other hand, P15 remarked, "The key to a good IoT experience is automation. Simply moving the control UI to my phone doesn't enable that". However, the same participant added, "Transferring a digital key to my phone for controlling a smart lock sounds very useful, since it would eliminate the need for a physical key and benefit both guests and hosts".
- *Pixelwheels obtained high usefulness and satisfaction scores despite low perceived performance.* P15 noted, "If I can use my smartphone as a controller, I could play console-like games on a smart TV without additional hardware, which is very useful". P8 and P14 echoed this view, saying, "Using the smartphone's sensors in combination with other systems feels very promising. The fact that COMET enables seamless access to not only sensors but also other hardware opens up a wide range of use cases".

In summary, our user study demonstrated that COMET is well-received across multiple apps in terms of performance, intuitiveness, usefulness, and satisfaction. The participants were able to complete multi-device tasks effectively and found that the component distribution interface was easy to use. Even in delay-sensitive scenarios such as Pixelwheels, they appreciated the utility of multi-device interaction, suggesting that COMET offers practical value despite moderate performance constraints.

12. Discussion

Feedback on multi-device app development. We conducted follow-up interviews with 10 participants who had prior experience in mobile app development to better understand their perspectives on building multi-device applications with COMET. Most participants (8 out of 10) cited high development complexity as the primary reason why multi-device apps remain rare in commercial app markets. They agreed that COMET significantly lowers this barrier by simplifying cross-device logic. While acknowledging this benefit, the remaining two participants raised concerns about the difficulty of testing and debugging distributed components, highlighting the need for dedicated developer tools.

API compatibility across devices. A practical challenge in component distribution is ensuring that the platform API version required by distributed components is supported on guest devices. To prevent failures, COMET embeds the host app's Android version constraints (e.g., `minSdkVersion`) into the manifest of the container sub-bundle at build time. Then, when a user initiates component distribution at runtime, COMET filters the list of paired devices to include only those that satisfy the version requirements, thereby ensuring compatibility.

In-execution component migration. The current prototype of COMET does not support the migration of a component to another device after instantiation, meaning that it must remain on the original device until execution is completed. Enabling such migration would enhance flexibility by allowing users to dynamically shift execution contexts across devices. However, this would require mechanisms to capture and transfer runtime state and then resume execution seamlessly, posing challenges such as preserving the in-memory state and re-establishing bindings to device-specific resources.

Limitations of static analysis. COMET uses build-time static analysis to collect dependencies for component distribution. However, dependencies introduced through reflection or dynamic loading may still be missed, which can lead to missing classes or resources in a distributed bundle. This limitation is not unique to COMET. Even in single-device apps, build-time optimizers or shrinkers may remove dynamically referenced code that is still required at runtime unless developers explicitly preserve it. To mitigate this issue, COMET provides an explicit override mechanism that allows developers to specify additional dependencies to retain (e.g., via R8/ProGuard keep rules), ensuring that hard-to-discover code paths and non-code metadata are included when needed. Finally, because COMET runs as a build-time plugin, it can introduce build overhead, particularly in clean builds (up to 2× in the worst case), although incremental builds can amortize this cost through Gradle caching.

Applying COMET to other mobile platforms. Because of its general design principles, COMET could potentially be applied to mobile platforms other than Android, such as iOS. COMET assumes that apps comprise modular components that can be dynamically launched and interact through local method calls. This design pattern is also commonly observed on other platforms. However, unlike Android, iOS prohibits such runtime behaviors under its current security constraints. If these constraints were relaxed, COMET could be adapted to support multi-device interaction on iOS. For example, distribution logic could be inserted during the build process using tools based on Swift Intermediate Language (Swift, 2025), and frameworks such as XPC (Apple, 2025) could be leveraged to implement cross-device RPC.

Failures in cross-device execution. As described in Section 8, COMET builds cross-device communication by extending Binder. As a result, COMET mostly follows Binder's failure behavior. If a remote endpoint fails during a call, the error is returned to the caller through Binder's normal exception mechanism instead of silently breaking the application's control flow. In our design, both cross-device method calls and synchronization-related RPCs (e.g., remote lock/unlock requests) are handled through dedicated Binder channels bridged by the daemons. Therefore, failures are reported in a way that is consistent with standard Android IPC. This preserves the expected exception behavior for RPC-like interactions.

Threat model and security considerations. Cross-device component distribution expands execution beyond a single device, which naturally creates more opportunities for attacks. We consider three main risks: (i) malicious peers (host or guest), (ii) tampered bundles, including warm-start caches, and (iii) interception or tampering on the communication path. To address these risks, the COMET daemon verifies trusted app provenance through package metadata such as the installer source and checks application signatures to detect tampered bundles or caches. Root detection can also be used to avoid relying on integrity checks in compromised environments. The communication channel can be further protected using secure transport such as TLS. In contrast, kernel-level compromise is not a threat specifically introduced by COMET through cross-device execution, but a broader issue that can also arise in the single-device setting. We therefore treat it as out of scope in this work.

Future work. Although COMET already demonstrates the feasibility of component-level multi-device collaboration, several important directions remain for future work.

First, improved developer tooling could make distributable apps easier to build and validate in practice. Useful extensions include better support for testing and debugging distributed components, as well as checks for annotation-related issues (e.g., missing required components or unnecessarily bundled ones) that may disrupt intended interaction flows or inflate bundle sizes.

Second, dependency-aware bundle composition could benefit resource-intensive apps. Distribution and installation overhead can increase for apps with complex dependencies, especially when overlapping libraries and resources are repeatedly included across multiple distributed components. A promising direction is to identify such shared dependencies at build time and package them as separate reusable sub-bundles that can be cached independently. This would reduce redundant transfers and help amortize installation costs across components and app sessions.

Third, we observed that multi-device collaboration can suffer not only from ordinary execution exceptions but also from additional runtime failures, such as device crashes and network disconnections. An important direction for future work is to improve robustness by supporting recovery of ongoing interactions while preserving the consistency of shared data across devices. This would require capturing and restoring relevant execution and application state after failures, and could also serve as a foundation for advanced extensions such as in-execution component migration.

13. Design comparison

Existing approaches to multi-device collaboration differ in both their design choices and the forms of interaction they support. To make these differences explicit and position COMET, we qualitatively analyze them along two key dimensions.

- **Applicability:** This dimension captures how readily an approach can be adopted in practice. We consider both *non-invasiveness* and *code transparency*. *Non-invasiveness* reflects whether an approach can be deployed on commodity devices without requiring system-level modifications (e.g., to the Android framework or kernel). *Code transparency* reflects the extent to which multi-device collaboration is supported at the system or framework level without requiring developers to implement additional communication logic within their applications.
- **Supported interaction types:** This dimension captures which types of multi-device collaboration each approach supports and how broad that support is. Building on the use case models introduced in Section 2, we examine whether a system supports *resource borrowing*, *service provision*, and *collaborative use*. This reflects whether an approach is limited to a specific collaboration pattern or can flexibly support diverse interaction scenarios across devices.

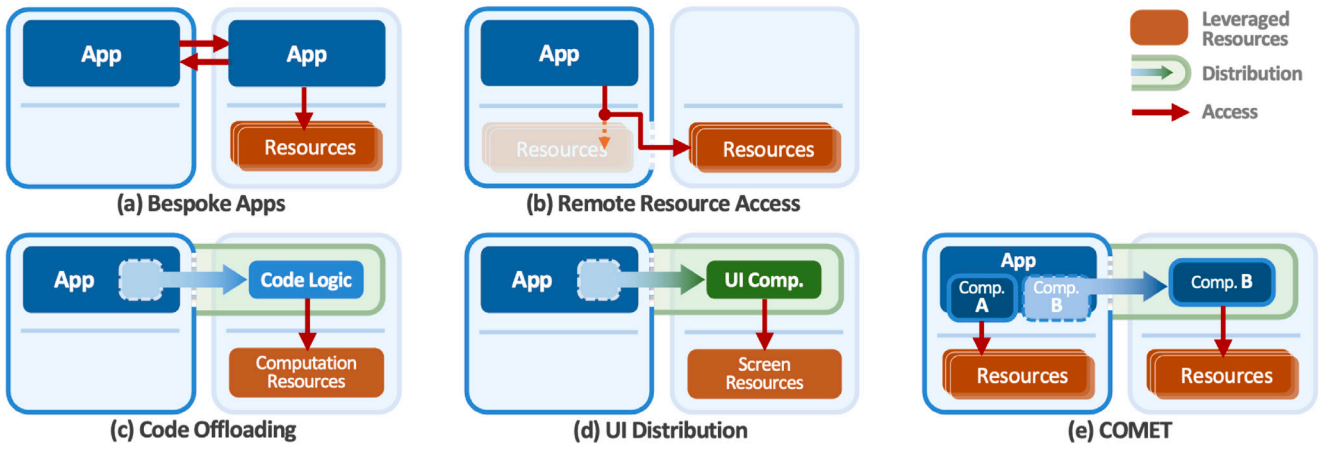


Fig. 12. Resource access and distribution units in representative multi-device collaboration approaches.

Table 3

Comparison of multi-device collaboration approaches across applicability and supported interaction types.

	Applicability		Supported interaction types		
	Non-invasiveness	Code transparency	Resource borrowing	Service provision	Collaborative Use
Bespoke Apps	O	X	Δ	X	Δ
Remote Resource Access	X	O	O	X	X
Code Offloading	X	O	Δ	X	X
UI Distribution	X	O	X	O	O
COMET (Proposed)	O	Δ	O	O	O

We use a three-level scale: O indicates support, Δ indicates partial or conditional support, and X indicates no support.

We review existing approaches along the two dimensions defined above to clarify how they differ in applicability and supported interaction types. Fig. 12 illustrates what each approach accesses or distributes across devices, and Table 3 summarizes the resulting differences. Here, we use the term resources broadly to refer to hardware resources, computational resources, and external application functionalities (e.g., SSO and mobile wallet services) that are available to an application. We evaluate each criterion at the approach level, determining whether a property is fundamentally supported and aligned with the primary design goals of that category.

Bespoke Apps. Bespoke applications implement multi-device support by explicitly embedding application-specific logic to leverage remote device resources. As shown in Fig. 12(a), this design typically requires corresponding applications on both devices, with developers manually implementing the entire cross-device communication, synchronization, and coordination flow. While it provides fine-grained control over user experience and interaction design, it incurs substantial engineering effort. As a result, bespoke applications can effectively support pre-designed multi-device interactions, but do not readily extend to new collaboration patterns.

Remote Resource Access. Remote resource access approaches, illustrated in Fig. 12(b), focus on enabling host applications to access resources on remote devices. Systems such as Rio (Amiri Sani et al., 2014) and MobilePlus (Oh et al., 2017) extend the I/O stack and Android RPC mechanisms so that existing applications can use remote resources with little or no application-level modification, thereby strongly supporting *resource borrowing*. However, because the host accesses only remote resources while keeping application logic and control flow local, these approaches do not support *service provision* and remain limited in the range of multi-device interaction types they support.

Code Offloading. In this category, code logic is the basic unit of distribution, as illustrated in Fig. 12(c). Systems such as DynaMix (Chae et al., 2018) support flexible computation placement by dynamically redistributing tasks across devices and integrating remote computation resources. Earlier work on code offloading (Cuervo et al., 2010; Chun

et al., 2011; Gordon et al., 2012) similarly focused on improving execution performance by relocating code logic to more capable devices. Accordingly, these approaches are effective for computation placement, but remain limited in supporting interaction types such as *service provision* and *collaborative use*, since they distribute code logic rather than user interaction or application functionality.

UI Distribution. Systems such as FLUID (Oh et al., 2019), FLUID-XP (Lee et al., 2021), and A-Mash (Lee et al., 2022) distribute fine-grained UI components across devices, enabling coordinated interaction with a single application, as illustrated in Fig. 12(d). In this way, the host can provide its application service through the guest device, which renders and interacts with the host application interface. As a result, these approaches support *service provision*. Moreover, by enabling two or more users to interact collaboratively across multiple devices, they also support *collaborative use* at the interface level. However, since these approaches distribute only UI components, they cannot directly support *resource borrowing* or other scenarios requiring access to guest-side device resources.

In summary, a common limitation across many prior approaches is that they rely on system-level modifications while still supporting only a limited range of interaction types. This is particularly limiting when a scenario spans more than one interaction type or resource requirement. For example, in the short-term letting scenario described in Section 2, UI distribution may be sufficient to let a guest interact with the host's IoT control interface. However, when the same scenario additionally requires access to guest-side device resources such as NFC for a digital room key, UI distribution alone cannot fully support such a scenario.

COMET distributes app components as the fundamental unit of execution. Because components encapsulate UI elements, resource access, and computation within a single unit, distributing them allows device resources to be flexibly utilized across devices for different collaboration scenarios. In this way, COMET supports diverse multi-device collaboration patterns while covering a broader range of scenarios explored in prior work. At the same time, COMET provides only partial code transparency. This is because developers must explicitly annotate

app components for distribution, which introduces some additional developer effort compared to fully transparent approaches. However, this design choice also allows developers to retain control over critical decisions, such as where sensitive logic executes and how interactions are structured across devices. This provides a practical balance between automation and developer control, reducing the need for additional cross-device implementation while preserving fine-grained control where it is needed.

14. Conclusion

We presented COMET, a novel mobile app framework that enables the dynamic distribution of app components across multiple devices. By enabling developers to annotate distributable components with minimal effort, COMET simplifies multi-device app development without requiring any system-level modifications. Furthermore, COMET maintains the illusion that distributed components are running locally within the host app, thus enabling seamless cross-device interaction without additional developer effort. Our evaluation shows that COMET not only lowers the development barrier of multi-device apps, but also delivers a responsive and satisfying user experience. We believe that COMET paves the way for the broader adoption of multi-device apps and creates new opportunities for richer cross-device experiences.

CRediT authorship contribution statement

Hyeonseok Yeom: Writing – original draft, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Hyosu Kim:** Writing – review & editing. **Steven Y. Ko:** Writing – review & editing, Supervision. **Young-Bae Ko:** Writing – review & editing, Supervision. **Sangeun Oh:** Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant (RS-2025-00522352, RS-2025-02217071), the Institute of Information & Communications Technology Planning & Evaluation (ITP) grants (IITP-2026-RS-2020-II201819, RS-2025-02305705, IITP-2026-RS-2023-00255968) funded by the Korea government (MSIT), and the Ajou University research fund.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at <https://doi.org/10.1016/j.jnca.2026.104513>.

Data availability

Data will be made available on request.

References

- AlDuaij, N., Nieh, J., 2021. Tap: an app framework for dynamically composable mobile systems. In: Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services. pp. 336–349. <http://dx.doi.org/10.1145/3458864.3467678>.
- AlDuaij, N., Van't Hof, A., Nieh, J., 2019. Heterogeneous multi-mobile computing. In: Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services. pp. 494–507. <http://dx.doi.org/10.1145/3307334.3326096>.
- Amiri Sani, A., Boos, K., Yun, M.H., Zhong, L., 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. pp. 259–272. <http://dx.doi.org/10.1145/2594368.2594370>.
- Apple, 2025. XPC: Access a low-level interprocess communication mechanism. <https://developer.apple.com/documentation/xpc>. (Accessed 24 January 2026).
- Card, S.K., 2018. The Psychology of Human-Computer Interaction. CRC Press.
- Chae, D., Kim, J., Lee, G., Kim, H., Chang, K.-A., Lee, H., Kim, J., 2018. {DynaMix}: Dynamic mobile device integration for efficient cross-device resource sharing. In: 2018 USENIX Annual Technical Conference. USENIX ATC 18, pp. 71–84.
- Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., Patti, A., 2011. Cloncloud: elastic execution between mobile device and cloud. In: Proceedings of the Sixth Conference on Computer Systems. pp. 301–314. <http://dx.doi.org/10.1145/1966445.1966473>.
- Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., Bahl, P., 2010. Maui: making smartphones last longer with code offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services. pp. 49–62. <http://dx.doi.org/10.1145/1814433.1814441>.
- Dean, J., Grove, D., Chambers, C., 1995. Optimization of object-oriented programs using static class hierarchy analysis. In: European Conference on Object-Oriented Programming. Springer, pp. 77–101. http://dx.doi.org/10.1007/3-540-49538-X_5.
- Google, 2025a. Cast from the YouTube app and YouTube.com. <https://support.google.com/chromecast/answer/2995235?hl=en>. (Accessed 24 January 2026).
- Google, 2025b. Google docs. <https://docs.google.com/>. (Accessed 24 January 2026).
- Google, 2025c. Overview of play feature delivery. <https://developer.android.com/guide/playcore/feature-delivery>. (Accessed 24 January 2026).
- Gordon, M.S., Jamshidi, D.A., Mahlke, S., Mao, Z.M., Chen, X., 2012. {COMET}: Code offload by migrating execution transparently. In: 10th USENIX Symposium on Operating Systems Design and Implementation. OSDI 12, pp. 93–106.
- Gradle, 2025. Understanding plugins. https://docs.gradle.org/current/userguide/custom_plugins.html#custom_plugins_2. (Accessed 24 January 2026).
- Ha, Y., Park, S., Lee, J., Oh, S., 2024. App-level multi-surface framework for supporting cross-platform user interface distribution. IEICE Trans. Inf. Syst. 107 (4), 564–568. <http://dx.doi.org/10.1587/transinf.2023EDL8060>.
- Kotlin, 2025a. Compiler plugins. <https://github.com/JetBrains/kotlin/blob/master/docs/compiler-plugins/basics.md>. (Accessed 24 January 2026).
- Kotlin, 2025b. FIR plugin API. <https://github.com/JetBrains/kotlin/blob/master/docs/fir/fir-plugins.md>. (Accessed 24 January 2026).
- Kotlin, 2025c. Kapt compiler plugin. <https://kotlinlang.org/docs/kapt.html#>. (Accessed 24 January 2026).
- Lee, S., Kim, H., Kim, S., Lee, S., Kim, H., Song, J.Y., Ko, S.Y., Oh, S., Shin, I., 2022. A-mash: providing single-app illusion for multi-app use through user-centric UI mashup. In: Proceedings of the 28th Annual International Conference on Mobile Computing and Networking. pp. 690–702. <http://dx.doi.org/10.1145/3495243.3560522>.
- Lee, S., Lee, H., Kim, H., Lee, S., Choi, J.W., Lee, Y., Lee, S., Kim, A., Song, J.Y., Oh, S., et al., 2021. FLUID-XP: Flexible user interface distribution for cross-platform experience. In: Proceedings of the 27th Annual International Conference on Mobile Computing and Networking. pp. 762–774. <http://dx.doi.org/10.1145/3447993.3483245>.
- Lee, J., Lee, H., Seo, B., Lee, Y.C., Han, H., Kang, S., 2018. SAMD: Fine-grained application sharing for mobile collaboration. In: 2018 IEEE International Conference on Pervasive Computing and Communications. PerCom, IEEE, pp. 1–10. <http://dx.doi.org/10.1109/PERCOM.2018.8444574>.
- Liang, S., 1999. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley Professional.
- Mozaffariahrar, E., Theoleyre, F., Menth, M., 2022. A survey of Wi-Fi 6: Technologies, advances, and challenges. Futur. Internet 14 (10), 293. <http://dx.doi.org/10.3390/fi14100293>.
- Netflix, 2025. How to use a mobile device to watch Netflix on a TV. <https://help.netflix.com/en/node/49>. (Accessed 24 January 2026).
- Nguyen, Q.-H., Blobel, J., Dressler, F., 2016. Energy consumption measurements as a basis for computational offloading for android smartphones. In: 2016 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC) and 15th International Symposium on Distributed Computing and Applications for Business Engineering. DCABES, IEEE, <http://dx.doi.org/10.1109/cse-euc-dcabs.2016.157>.
- Nielsen, J., 1994. U Sability Engineering. Morgan Kaufmann.
- Oh, S., Kim, A., Lee, S., Lee, K., Jeong, D.R., Ko, S.Y., Shin, I., 2019. FLUID: Flexible user interface distribution for ubiquitous multi-device interaction. In: The 25th Annual International Conference on Mobile Computing and Networking. pp. 1–16. <http://dx.doi.org/10.1145/3300061.3345443>.

- Oh, S., Kim, A., Lee, S., Lee, K., Jeong, D.R., Ko, S.Y., Shin, I., 2024. Supporting flexible and transparent user interface distribution across mobile devices. *IEEE Trans. Mob. Comput.* 23 (8), 8398–8417. <http://dx.doi.org/10.1109/tmc.2024.3349561>.
- Oh, S., Yoo, H., Jeong, D.R., Bui, D.H., Shin, I., 2017. Mobile plus: Multi-device mobile platform for cross-device functionality sharing. In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. pp. 332–344. <http://dx.doi.org/10.1145/3081333.3081348>.
- Perez, S., 2018. Nielsen: the second screen is booming as 45% often or always use devices while watching TV. <https://techcrunch.com/2018/12/12/nielsen-the-second-screen-is-booming-as-45-often-or-always-use-devices-while-watching-tv/>. (Accessed 24 January 2026).
- research, P.A., 2024. At CES® 2024, parks associates announces new research showing average number of connected devices per US internet household reached 17 in 2023. <https://www.parksassociates.com/blogs/press-releases/at-ces-2024-parks-associates-announces-new-research-showing-average-number-of-connected-devices-per-us-internet-household-reached-17-in-2023>. (Accessed 24 January 2026).
- Shen, G., Li, Y., Zhang, Y., 2007. Mobius: enable together-viewing video experience across two mobile devices. In: *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*. pp. 30–42. <http://dx.doi.org/10.1145/1247660.1247667>.
- Steam, 2025. Steam remote play. <https://store.steampowered.com/remoteplay/>. (Accessed 24 January 2026).
- Steensgaard, B., 1996. Points-to analysis in almost linear time. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 32–41. <http://dx.doi.org/10.1145/237721.237727>.
- Swift, 2025. SIL introduction. <https://github.com/swiftlang/swift/blob/main/docs/SIL/SIL.md>. (Accessed 24 January 2026).
- Van't Hof, A., Jamjoom, H., Nieh, J., Williams, D., 2015. Flux: Multi-surface computing in android. In: *Proceedings of the Tenth European Conference on Computer Systems*. pp. 1–17. <http://dx.doi.org/10.1145/2741948.2741955>.
- Xu, J., Cao, Q., Prakash, A., Balasubramanian, A., Porter, D.E., 2017. UIWear: Easily adapting user interfaces for wearable devices. In: *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. pp. 369–382. <http://dx.doi.org/10.1145/3117811.3117819>.
- Zheng, J., Peng, X., Yang, J., Cai, H., Huang, G., Zhang, Y., Zhao, W., 2017. CollaDroid: Automatic augmentation of android application with lightweight interactive collaboration. In: *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. pp. 2462–2474. <http://dx.doi.org/10.1145/2998181.2998278>.



Hyeonseok Yeom received the B.S. degree in digital media, software and computer engineering (double major) from Ajou University, Suwon, South Korea, in 2022. He is currently pursuing the Ph.D. degree in the Department of Artificial Intelligence Convergence Network at Ajou University. His research interests include mobile computing and multi-device experiences.



Hyosu Kim received a B.S. degree from the Department of Computer Engineering at Sungkyunkwan University in 2010. He received M.S. and Ph.D. degrees from the School of Computing at Korean Advanced Institute of Science and Technology (KAIST), Republic of Korea in 2012 and 2018, respectively. He is currently an associate professor at Chung-Ang University, School of Computer Science and Engineering, Seoul, South Korea. His research interests include cyber-physical systems, mobile and ubiquitous systems, and privacy.



Steven Y. Ko is an Associate Professor in the School of Computing Science at Simon Fraser University. His current research interest is improving the reliability and security of mobile systems by developing techniques at the intersection of mobile systems and software engineering. He received a B.S. degree in Mathematics from Yonsei University, an MS in Computer Science and Engineering from Seoul National University, and a PhD in Computer Science from the University of Illinois at Urbana-Champaign. He worked as a postdoc in the Department of Computer Science at Princeton University. Until 2020, He was with the Department of Computer Science and Engineering at the University at Buffalo, The State University of New York as an Associate Professor.



Young-Bae Ko received the Ph.D. degree in computer science from Texas A&M University, College Station, TX, USA, in 2000. From 2000 to 2002, he was a Research Staff Member with the IBM Thomas J. Watson Research Center, NY, USA. Since 2002, he has been with Ajou University, where he has been conducting research in the areas of mobile computing and wireless networking. He is currently a Professor with the Department of Software/AI Convergence Networks, Ajou University, South Korea. He is leading the Brain Korea 21 FOUR Research and Development Group for 6G at Ajou University supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT(MSIT). His recent research interests include mobile computing, high-precision positioning and object tracking based on UWB and AI, seamless handover in cellular-V2X, and AI-empowered IoT (AIoT) systems.



Sangeun Oh is an Assistant Professor with the Department of Computer Science and Engineering, Korea University, Seoul, South Korea. His research interests include mobile/IoT systems and real-time embedded systems. He received the B.S. degree in computer and communication engineering from Korea University, Seoul, South Korea in 2012, and the M.S. and Ph.D. degrees in computer science from KAIST, Daejeon, South Korea, in 2014 and 2020, respectively. Until August 2024, he was an Assistant Professor with the Department of Software and Computer Engineering, Ajou University, Suwon, South Korea.