Le *et al. Hum. Cent. Comput. Inf. Sci. (2019) 9:20*
https://doi.org/10.1186/s13673-019-0182-9

Human-centric Computing and Information Sciences

RESEARCH Open Access

Enabling multi-hop remote method invocation in device-to-device networks

Minh Le[1], Stephen Clyde[1] and Young-Woo Kwon[2*]

*Correspondence: ywkwon@knu.ac.kr
[2] Department of Computer Science and Engineering, Kyungpook National University, Daegu, South Korea
Full list of author information is available at the end of the article

Abstract
To avoid shrinking down the performance and preserve energy, low-end mobile devices can collaborate with the nearby ones by offloading computation intensive code. However, despite the long research history, code offloading is dilatory and unfit for applications that require rapidly consecutive requests per short period. Even though Remote Procedure Call (RPC) is apparently one possible approach that can address this problem, the RPC-based or message queue-based techniques are obsolete or unwieldy for mobile platforms. Moreover, the need of accessibility beyond the limit reach of the device-to-device (D2D) networks originates another problem. This article introduces a new software framework to overcome these shortcomings by enabling routing RPC architecture on multiple group device-to-device networks. Our framework provides annotations for declaring distribution decision and out-of-box components that enable peer-to-peer offloading, even when a client app and the service provider do not have a direct network link or Internet connectivity. This article also discusses the two typical mobile applications that built on top of the framework for chatting and remote browsing services, as well as the empirical experiments with actual test-bed devices to unveil the low overhead conduct and similar performance as RPC in reality.

Keywords: Android RPC, Method invocation routing, WiFi-Direct, Group-to-group, Mobile network

Introduction

Resource sharing or computation offloading on mobile networks can bring a lot of benefits, the approaches in this domain if possible, can be widely applied to IoT networks or ubiquitous cities. In this section, we will go through the several limitations of current technologies, as well as our motivation to build a middleware that overcomes these obstacles by extending Remote Method Invocation method and multi-hop capability to enable resource sharing over mobile networks.

Low-end devices always have trouble running intensive resource-consuming applications such as image or video processing, which remarkably slow down its speed and drain energy. One well-known solution is having the device participate in a collaboration in which it can offload or migrate intensive code portions [1–3] onto another device or cloud server with copious resource capacity, have them execute the code and wait for responses [4, 5]. Although the idea is straightforward, code offloading has not been widely applied in the mobile industry due to several issues: it requires radical changes to the system core (e.g. OS rooting) and originates high latency, making it inapplicable for the category of applications that send a huge number of requests in a short period (e.g. real-time).

Remote Method Invocation (RMI) is a distributed architecture in which methods of remote Java objects can be invoked from other Java virtual machines possibly located on different hosts [6]. One of its strengths is that it provides a degree of location transparency by having servers add services to a registry and requiring clients to use the registry for binding. Having RMI enabled on mobile platforms can bring several benefits, which the best of those is *device resource transparency* when multiple resources can be allocated for one function call. However, the original RMI technology does not support any routing other than what the underlying network layers support, if there is no inter-network (network-layer routing) between two devices, an object on the first device cannot invoke a method for an object on the second device. Moreover, RMI is an obsolete technology that heavily relies on the **javax** package from the Java SDK library; therefore, it is not supported on mobile platforms like Android.

Technology based on object brokers, like CORBA [7], can support remote method invocations. However, they rely on middleware processes (or threads) to instantiate or re-hydrate objects and then bind method calls to the target objects. Although there have been attempts to support CORBA on mobile platforms [8], they require the underlying layers to handle inter-networking and therefore do not directly support inter-group communications. In addition, the authors believe that its middleware is too heavy for most mobile devices and that its language-neutral approach to distribution is unnecessarily complex for most mobile apps.

RabbitMQ [9] or ActiveMQ [10] or the other current message queues are the fully implemented middleware that is widely used in many different research and commercial products; they can be deployed in distributed and federated configurations to support high-scale and high-availability software architectures. However, the main obstacle of these popular middleware systems is that the central server application must be located on a stationary server not a mobile device because it requires considerable system resources and platform dependencies [11, 12], and thus the entire system is unable to deploy on a self-operating mobile network. Likewise, the same problem also occurs for other publish-subscribe middleware systems in which a developer needs to become familiar with the libraries that always overwhelm the mobile platforms.

Most of the new generation phones feature closed-range, non-Internet communications such as Bluetooth, NFC and WiFi-Direct [13]. However, while WiFi-Direct can allow handshake between two devices that are nearly 200 m apart, Bluetooth and NFC can only work at small distances. While Bluetooth and NFC can only pair between two devices, WiFi-Direct enables connections between an unlimited number of devices as long as they are within the supported distance range. A WiFi-Direct network is a client-server model in which one device is elected to become a group owner, and the others connect to the owner as the clients. According to this model, once a device joined a group, it is by default unable to be contacted by any other groups. There are some solutions to address this limitation [5], but, these solutions lack a standard software model or out-of-the-box library to quickly develop or integrate into a software system.

To simplify the networking development of mobile applications and extend the limited range of non-Internet communications such as WiFi-Direct, we introduce our new middleware system that enables remote method execution routing on device-to-device networks (D2D). Our library adopts a group-to-group network architecture that can extend the range of communication, so that a device from one group can talk to a device from another group through a virtual bridge. Firstly, the developer implements functions and declares annotations on those he/she wants to publish to define the service. Then, during the code compilation, the compiler will automatically generate the software components that will be used to construct D2D networks. By integrating these constituent components into the app, a user can initiate a network in any topology since these components can flexibly switch between the devices. In "Evaluation" section we will demonstrate the use of the software library to quickly build two applications, *chat* and *remote browser*, that enable communications over multi-group device-to-device network. In this article, we introduce our new middleware architecture that makes the following contributions:

- *A flexible middleware architecture* that is easy to use, and able to address multi-hop D2D communications, as well as extendable to group-to-server, as long as the network is available.
- *A new routing mechanism to enable mobile RMI* on both mobile and stationary server platforms, adapted to the group-to-group communication through annotations.
- *Introduction of real applications and empirical experiments* with an actual-device testbed through benchmarks and use cases to demonstrate the performance of our middleware compared to the others.

Background

Although the research aims to extend the communication capacities of mobile devices in all cases, this article only focuses on one network interface that is WiFi-Direct because of its long range distance and availability (whenever a Wi-Fi network is active). We believe the same system architecture can also be applied to the other network interfaces on the same device such as NFC or Bluetooth.

WiFi-Direct

Wi-Fi Direct is a new peer-to-peer (P2P) communication standard built on top of the IEEE 802.11 to provide direct connections between Wi-Fi-enabled devices without Internet connections [14]. In our prior research [4, 15, 16], we used Wi-Fi Direct to construct P2P networks (i.e., groups) among the nearby devices, by letting them dynamically discover and connect to each other. However, with WiFi Direct, a single device can only belong to a single group at any time. It is still possible, though, for a device to use its legacy WiFi client (LC) to connect to an Internet access point or any other peer device directly.

Group-to-group in WiFi-Direct

We discuss a solution to overcome the range limitation when executing mobile services in a P2P network without an Internet connection. In short-range P2P networks formed

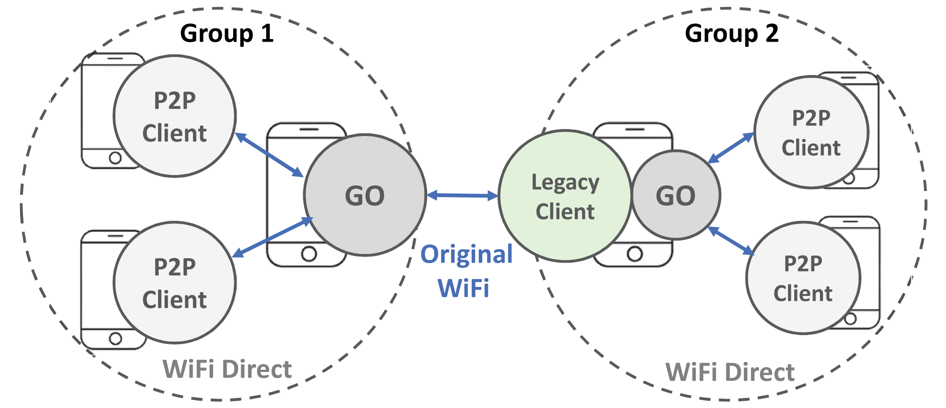


Fig. 1 Use of WiFi interface to create group-to-group communications

by Wi-Fi-Direct, all connecting devices are summoned into one group led by a Group Owner (GO). When the first two devices start the communication via Wi-Fi Direct, the protocol will evaluate and designate one device to be a GO and the other will become a client. These devices will be connected via the Client-Server model. When another device joins the network, it is connect to the GO server as the client, and thus all the participating devices form up a Wi-Fi Direct group with one GO working as the server. Among the groups, the GOs from different groups are unable to connect to each other, and so are the clients. This shortcoming makes it impossible for the devices to communicate over the distance of 200 m, which is the maximum range of Wi-Fi Direct. To overcome this limitation, we leverage the original WiFi interface of the GO and make it a *bridge*, or *Legacy Client* [14], to another group (Fig. 1).

When one device becomes GO, its virtual access point (i.e., soft AP) runs a DHCP service to automatically assign private IP addresses (e.g., 192.168.xxx.yyy) to itself and other clients its group. Being a GO, the device is also exposed to the others as a *WiFi access point*, so that any nearby devices can find and connect to it on the *original interface*. Therefore, a user can create an Legacy Client (LC) on the GO of another group to connect to that GO through the *original WiFi interface*. After the LC is connected, it is assigned an IP in the same range of the assigned IP address (e.g., 192.168.xxx.yyy). A similar approach has been successfully applied in the content-centric routing domain [17]. Finally, in this research, we assume that a mobile network is already formed up before an application starts so as to focus on the application layer to evaluate our new middleware architecture [18, 19].

Related work

Our system performs RMI by serializing a function call into a binary stream and dispatching over a wireless network. In the same domain, Android RMI [20] leverages the original Binder to allow users to invoke system services as well as application services between devices using a remote parcel format. Lin et al. introduces a cross-platform IPC mechanism called XBinder [21] to enable remote processes among multi-user communication for mobile applications to cooperate with local or remote services without forming a complicate network. However, despite the remarkable improvements with respect to performance [22], these design choices only target mobile devices connected in the same network, and thus they will not work when devices are moving to the other networks. To address this issue, we adopted a group-to-group network architecture to help flexibly switch the roles of devices during run-time and allow reconfiguring the network in multiple topologies.

Nakao et al. [23] provides an RPC-based invocation mechanism between Android devices using Intent, a message format used by the Android platform to realize transparent remote service communications to other devices without any modifications to the existing Android applications. Similarly, Nagahara et al. [24] proposed a distributed intent framework in which Android applications collaborate with embedded devices by sending serialized Intent messages through the network. Another approach for mobile remote processes is making services public, so that other devices may invoke services using remote call mechanisms [25, 26], but this approach incurs too much overhead for the host device as well as posing risks of unavailability of services when the host moves out of the communication range.

Our middleware contributes to the domain of WiFi-Direct multi-group communication in which one group connects to another using a legacy client and a special component operating on the original Wi-Fi interface to serve as a bridge between the two group owners [14, 27, 28]. Casetti et al. [17] leverages Wi-Fi Direct to support multiple groups for a content-centric routing network in which data is transparently available to users using content routing tables that collect and transport data over the content nodes. Before the execution, routing tables are advertised and populated via a registration/advertisement protocol. Our system extends the idea of the content-centric network to bring the function-centric mobile network, in which any device can request for a function call regardless of knowing the actual location of the function (i.e., the requested function can be hosted on a certain mobile device or a stationary server).

In the category of the wireless P2P communication before the Wi-Fi Direct technology, several efforts utilized wireless communications in an add-hoc fashion to establish P2P networks, such as media sharing systems on urban transportation using Bluetooth [29], resource sharing using cellular networks [30], and radio resource sharing over ultra-wideband [31]. Built on top of Wi-Fi P2P, Rio [32] leverages I/O devices to capture and share contents and resources between the existing applications running on different devices without any modification. Some of its applications are multi-system photography and gaming, music and video sharing, and SIM card sharing for multiple devices. GameOn [33] was also built on the same network infrastructure to establish non-Internet connection between gamers within closed range networks like on public transportation. CAMEO [34], and GigaSight [35] are also the similar content sharing systems in closed range network architectures.

Finally, although we used real devices and environments for our experiments, there is a large gap between testing devices and the real world that contains hundreds of devices and different situations. Therefore, we share the same vision with the network simulation research, especially for multi-group WiFi-Direct networks to overcome following two issues: (1) the high cost of the experiment deployment with a vast number of devices and (2) the complexity of the network discovery and handshake phases. WiDiSi [6] is a dedicated visual simulation extending the PeerSim library [36] to support WiFi-Direct, it can simulate and visualize a vast D2D network including the discovery and network establishment of devices moving randomly within closed distances. However, the disadvantages of WiDiSi as well as PeerSim are being single-threaded, having less autonomy and unsupported not supporting multiple groups. The new result of that work, called MAGNET [8], is a novel self-organizing middleware infrastructure that aims to provide reliable and stable P2P connectivity among large numbers of smart devices.

Approach

We built our middleware on top of the ZeroMQ (ZMQ) library [37], a flexible library for message queues which is available on multiple platforms including Android. As a result, our first version works on both Android and PC systems. Our middleware system can be simply employed via two steps: a developer creates the service with full implementation and marks our service annotations (Code Snippet 4). After compiling the project, the middleware's processor will automatically generate extension classes for the service; then, the developer uses these classes along with the other basic components such as `Broker` and `Bridge` to construct their mobile networks (Code Snippet 5).

Middleware components

The middleware consists of six main components: the `Broker`, `Worker`, `Client`, `Requester`, `Responder` and `Bridge` component; each has different functionality but shares the same basic structure including ring buffers for incoming and outgoing messages.

Figure 2 depicts the potential communications among the components. In the first type, the `Worker` connects and registers its services to the `Broker` while the `Broker` buffers the request messages sent from the `Client`, forwards each request to the corresponding `Worker` to resolve and forwards result back to the `Client`. The fourth type introduces a more sophisticated strategy with the involvement of a `Bridge`, an intermediate between two `Brokers`. The `Bridge` comprises of a `Client` and `Worker`; one connects to the left `Broker` and the other connects to the right. These two types will be used for *Peer-to-Peer* and *Group-to-Group* modes.

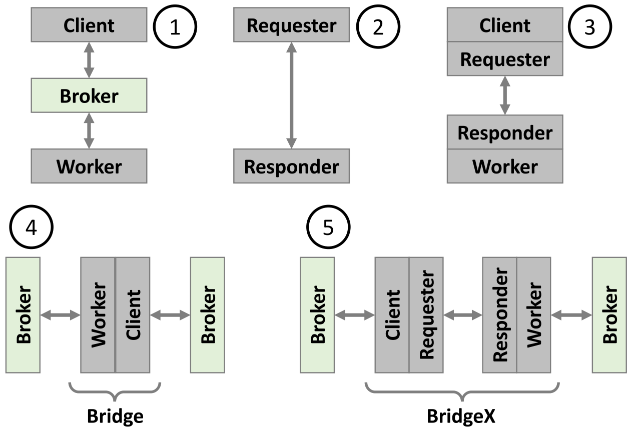


Fig. 2 All middleware components

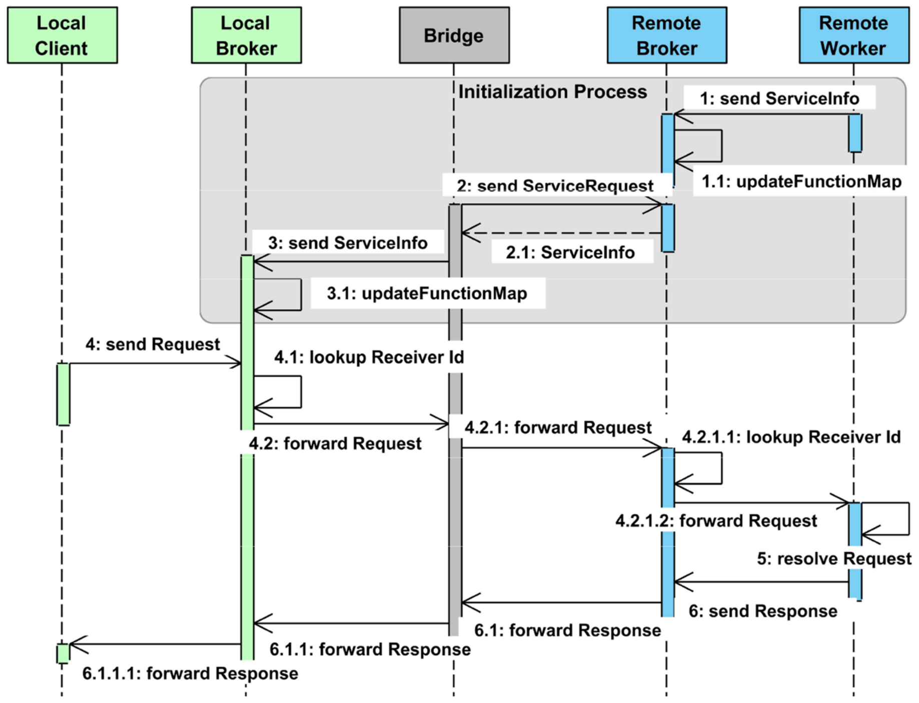


Fig. 3 Sequences of the initialization process and message requests

The `Bridge` is simply a forwarder that starts after the `Workers` are settled. Firstly, it sends a *Service Request* to the remote `Broker` to retrieve the list of available `Worker` services. Then, it connects to the local `Broker` and registers the remote services as it is going to forward, the local `Broker` will register those services under the `Bridge`'s ID.

```
{ "code":"REGISTER",
  "id":"1",
  "functions":[
    { "functionName":"greeting",
      "inParams":["String"],
      "outParam":"String[]"},
    { "functionName":"
        getFolderList",
      "inParams":["String"],
      "outParam":"String[]"}]}
```

Code Snippet 1: Worker's service definition in JSON format.

The second type involves a `Requester` and a `Responder`; one sends a request and the other responds in the *synchronous mode* like in the *Client-Server* model. One extension of this type called `BridgeX` which involves a `Client` and `Requester` on one side and a `Responder` and `Worker` on the other side. In reality, while the `Bridge` model needs four steps to establish a connection between two Brokers, `BridgeX` needs just three steps. The detailed usage of `Bridge` and `BridgeX` will be discussed in "Group-to-group communication" section (Figs. 6, 7).

These components do not start at the same time. Generally, `Broker` always starts first, right after network establishment to either host services for its current group or interface with the other groups. `Workers` start after the `Broker` to register their services, when it starts, it sends the service definition in JSON format (Code Snippet 1) to the `Broker`, the `Broker` will extract the function list and store them in its `FunctionMap` table where *keys* are *function names* and values are *Worker IDs*. Later, the `Broker` will use `FuncName` from a request message to find the according `Worker` and forward the request. When `Worker` leaves the network, it sends the `Broker` an instruction message with code `UNREGISTER` to remove all of its services from the `Broker`'s group map. Figure 3 describes the sequences of the initialization process and message requests between a client and remote workers.

In our library, only `Broker` and `Bridge` from the other software are used in their original forms original forms. To generate the other components, developers have to follow their interface prototypes to define the implementation. These components will be created during the code compilation by annotations.

Function calls to messages

The middleware serializes a function call into a request message and dispatches the request to an appropriate device. The `RequestMessage` object holds request content that was sent from the Client. To this end, it has `functionName` to keep the name of the function, `InParams` to contain types and values of input parameters and `OutParam` to describe the type of output parameter; the parameter type can be a single value or an array of primitives or any user-defined object, as long as the relative classes exist in the *classpath* during the compilation and execution on all sides.

During the compilation, the `AnnotationProcessor` examines the service function prototypes marked with `ServiceMethod` annotations and generates the Client class. The processor automatically fills each function with three different portions: (1) create a `RequestMessage` to wrap up input and output parameters of the function, (2) serialize the request to binary data and (3) use the default `send` function to dispatch the binary message to the Broker (see Code Snippet 2).

```
public void getFolderList(String path) {
  String functionName = "getFolderList";
  String outType = "java.lang.String[]";
  RequestMessage reqMsg = new RequestMessage(
                      functionName, outType);
  reqMsg.inParams = new InParam[1];
  reqMsg.inParams[0] = new InParam("path",
                      "java.lang.String", path);
  byte[] reqBytes = NetUtils.serialize(reqMsg);
  this.client.send(functionName, reqBytes);
}
```

Code Snippet 2: Function in `Client` class for the service in Code Snippet 4.

At the generated Worker, each request is deserialized to a Java object and categorized by `functionName`. Inside each *method handler*, input parameters collected from the `RequestMessage` are passed to the actual function call of the service instance with the output type is defined by `OutParam`. Finally, the result of the call is wrapped within a `ResponseMessage` along with the name and type and is sent back to the Broker (Code Snippet 3).

```
switch (functionName) {
[...]
case "getFolderList": {
  /* variable "path" */
  String[] paths = new String[req.inParams[0].values.length];
  for (int i = 0; i < req.inParams[0].values.length; i++)
    paths[i] = (String) req.inParams[0].values[i];
  String path = paths[0];
  /* start calling function "getFolderList" */
  String[] rets = serviceA.getFolderList(path);
  String retType = "String[]";
  ResponseMessage resp = new ResponseMessage(req.messageId,
                            req.functionName, retType, rets);
  /* convert to binary array */
  respBytes = NetUtils.serialize(resp);
  break;
}
```

Code Snippet 3: Function in `Worker` class for the service in Code Snippet 4.

Message flows

In this section, we describe the design of low-level message flows on top of ZMQ from Client to Worker through Brokers and Bridges and vise versa. In ZMQ, a message traveling between the two sockets needs at least two parameters: the *identity of the destination* and the *message content*. To avoid overheads of message transit on the intermediates, we design message format with the following fields: `ReceiverID`—the identity of the destination, `ClientIDs`—the ID chain of Clients, `FuncName` and `Message`—a serialized Message object. Specifically, `ClientIDs` keeps a series of Client IDs which it passes along to the Worker. For example, in Fig. 4, when the message arrives at the Worker the value of `ClientIDs` is "`1/100/200`" where 1 is the ID of Client #1, 100 is the ID of the Bridge's Client #1 and 200 is the ID of Bridge's Client #2. `ClientIds` is filled during the request process and consumed in the response.

Sending a request

We describe the Request Flow using a typical example in Fig. 4: a message to a Broker does not need an address, so the first message's `ReceiverID` is `EMPTY` and `ClientIDs` is "1," since the message came out from Client with ID is 1. When Broker receives the request, it looks up `FuncName` in the `FunctionMap` to find the relative Bridge and forwards the message. The Bridge concatenates `ClientIDs` with the ID of its Client and forwards the request to the next Broker. This process repeats until the request eventually meets a Worker and gets resolved. If for any reasons the request cannot find a Worker, a denial message with the flag `WOKRER_NOT_FOUND` will be sent back to the Client as a response.

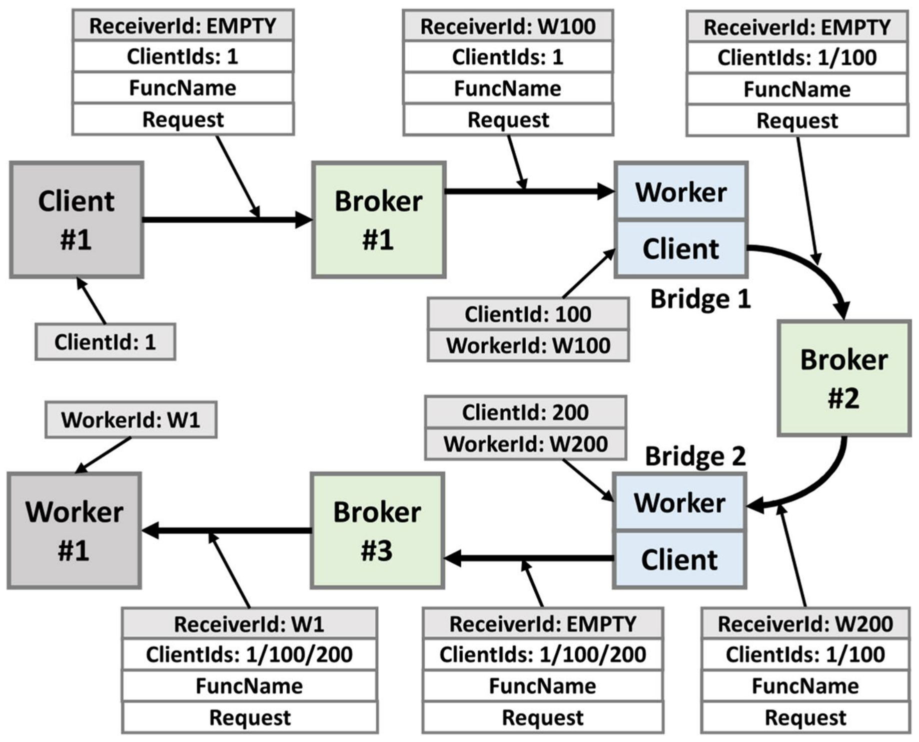


Fig. 4 Message flow from a `Client` to the a `Worker`

Sending a response

Figure 5 illustrates a return flow from the Worker to the requesting Client. When the response arrives at the Broker, the Broker will extract the first ID in the `ClientIDs` and put it to the `ReceiverID` so that the response can find the next destination. This process repeats until the `ClientIDs` is `EMPTY`, in other words the response arrives at the requesting Client. If for any reasons the response can't find the way back to the Client (when `ReceiverID` not found or `ClientIDs` is `EMPTY`), the Client will wait until timeout to report an `UNAVAILABLE_SERVICE` error.

Service definition

A developer indicates a class as a service by declaring the `@MobileService` annotation at the class scope (Code Snippet 4). We support two communication models: *Client-Server*—using Requester and Responder objects, and *P2P*—using Client and Worker objects, defined by the `commModel` option. A user can change `transmitType` to switch transmission type to either binary or JSON format, the default value is `TransmitType.Binary`.

A function is part of a service if it comes with the `@ServiceMethod` annotation; those without this annotation will be excluded. The developer can choose `syncMode` to be either `Async` or `Sync`. In the case of `Sync`, the requesting Client will wait until the arrival of the response or timeout to end the transaction. The last parameter `suffix` annotates the indexes of the overload functions.

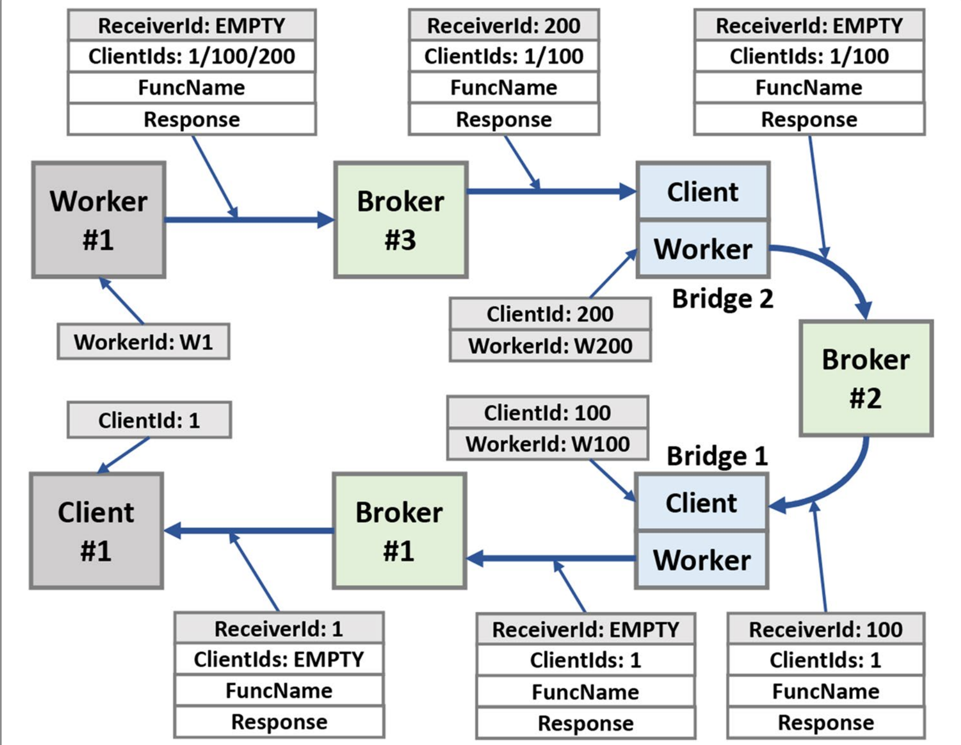


Fig. 5 Message flow from a `Worker` back to the `Client`

```
@MobileService(
        commModel = CommModel.P2P,
        transmitType = TransmitType.Binary)
public class ServiceA {
    @ServiceMethod(syncMode = SyncMode.Async)
    public String[] greeting(String msg) {
        return new String[] { msg, msg.toUpperCase()
             };
    }
    @ServiceMethod(
            syncMode = SyncMode.Async,
            suffix = "2")
    public String[] getFolderList(String path) {
        File folder = new File(path);
        File[] files = folder.listFiles();
        String[] res = new String[files.length];
        for (int i = 0; i < files.length; i++)
            res[i] = files[i].getAbsolutePath();
        return res;
    }
}
```

Code Snippet 4: Service definition example.

When the developer compiles the code, the Annotation Processor will automatically generate Client and Worker objects for the service with additional suffixes, for example `ServiceAClient` and `ServiceAWorker` for the `ServiceA` service (see "Function calls to messages" section). The developer can utilize these objects to construct a mobile network along with the Broker and Bridge in many ways. the Code Snippet 5 shows an example with two Brokers on two different devices, one middle Bridge, a Client and a Worker. When implementing the Client code, since `syncMode` is `Async`, a developer needs to override the `received()` method to handle incoming responses, so as to check the responses with label `BROKER_INFO`. In addition, error messages returning from the Broker should be handled in this method.

```
/* start a Broker and a Worker on the remote device */
new Broker(remoteBrokerIp, clientPort, workerPort);
new ServiceAWorker(remoteBrokerIp, workerPort);
[...]
/* start a Broker and a Worker on local device */
new Broker(localBrokerIp, clientPort, workerPort);
/* start a Bridge to bridge between the local and remote Brokers */
new Bridge(localBrokerIp, workerPort, remoteBrokerIp, clientPort);
/* start a Client at local */
ServiceAClient client = new ServiceAClient(localBrokerIp,
                        clientPort, new ReceiveListener() {
  @Override
  public void received(String idChain, String funcName,
                              byte[] data){
    ResponseMessage resp = NetUtils.deserialize(data);
    if (resp.functionName.equals(NetUtils.BROKER_INFO)){
      /* a denied message from the Broker */
      Log.v("Error: " + resp.outParam.values[0]);
    }else if (resp.functionName.equals("greeting")){
      /* results from the "greeting" function */
      Log.v("Received: " + resp.outParam.values[0]);
    }else if (resp.functionName.equals("getFolderList")){
      /* results from the "getFolderList" function */
      String[] files = (String[]) resp.outParam.values;
      for (int i = 0; i < files.length; i++)
        Log.v("File: " + files[i]);
    }
}});
client.getFolderList("/");
```

Code Snippet 5: Adopt components and construct a mobile network.

Group-to-group communications

In the previous section, we discussed the idea of leveraging the original Wi-Fi interface to enable group-to-group communication. In this section we will detail the deployment of the middleware. Figure 6 illustrates a typical case of two groups 1 and 2, in which each group has two devices: one takes the role of GO with a Broker and another starts a Client.

To implement a *LC*, we first let the Group 1's GO connect to the Wi-Fi Access Point (AP) created by Group 2's GO. As aforementioned, when a device becomes a GO, it also becomes a WiFi AP, and the other devices can connect to it via the Wi-Fi interface. Then, we start a new Broker on the Group 2's GO to host on the IP address of the Wi-Fi interface, which is completely irrelevant to the Wi-Fi Direct network established before. A new Bridge will start on the Group 2's GO to connect the two Brokers on Group 2, in the meantime, a new Bridge will start on the Group 1's GO to connect its Broker with the one on Group 2's GO over the Wi-Fi interface (Fig. 6). From this moment, the system operates exactly the same way as the one described in "Message flows" section.

To simplify the complexity of the system, we can also use `BridgeX` to replace one Broker on Group 2's GO device (Fig. 7), so that the two `BridgeXs` on each group will establish a pair connection. As we described before, our system can work on both Android and PC platforms, the developer can easily bridge a communication from a device to PC by deploying a Broker on a PC to host the connections from devices (Fig. 8). On a mobile device, we can use a Bridge to relay messages back and forth from mobile Brokers to the PC.

Applicability

Since the middleware system addresses the issue of extending the limited communication range of mobile networks, especially when there is no need of Internet, it can be used across many different applications. In this section, to show the ease of a software development, we further discuss two mobile applications that utilize the proposed middleware system.

Remote browser

In this application, a user can access WWW without an Internet connection. In order to realize this idea, we developed a simple function `getUrl` to download contents of an URL and return in binary format, using `OkHttp` library.[1]

```
public byte[] getUrl(String url) {
  if (client == null) {
    client = new OkHttpClient.Builder().build();
  }
  try {
    Request request = new Request.Builder().url(url).build();
    Response response = client.newCall(request).execute();
    if (!response.isSuccessful()) {
      throw new IOException("unexpected code " + response);
    }
    return reponse.body().bytes();
  } catch(IOException e) {
    return new byte[0];
  }
}
```

Code Snippet 6: Sample code for **`getUrl`** function.

To load webpages, a browser first calls `getUrl()` to download the contents of the page and find the inner links which could be other HTML pages, CSS or JavaScript or

[1] `OkHttp`: http://square.github.io/okhttp.

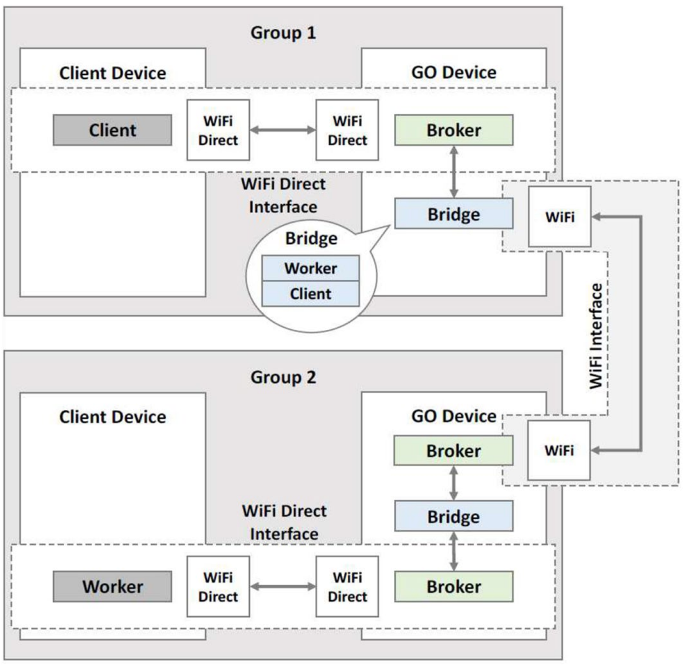


Fig. 6 Multi-group communication with 4 devices using `Bridge`

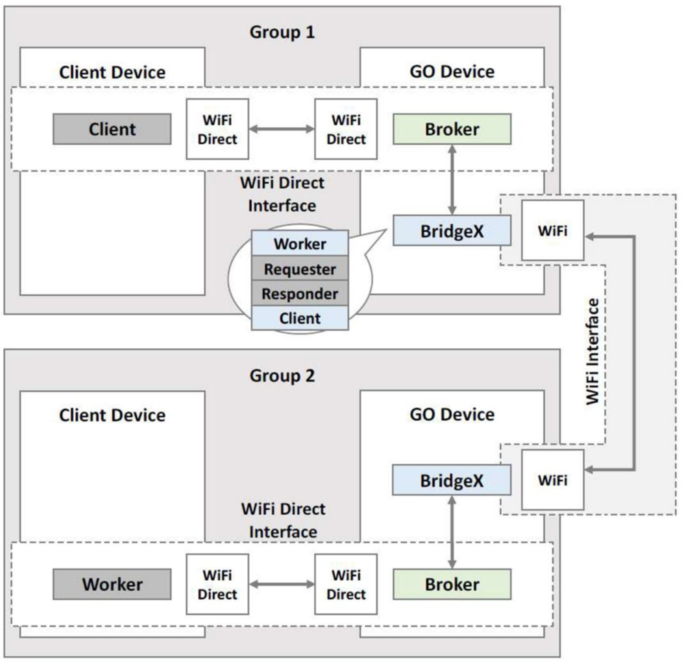


Fig. 7 Multi-group communication with 4 devices using `BridgeX`

multimedia resources. Then the browser recursively downloads the contents of the inner links using that `getUrl` function. To automatically fetch an URL this way, we use the `NanoHttpD` library as the local web server to handle loading and extracting

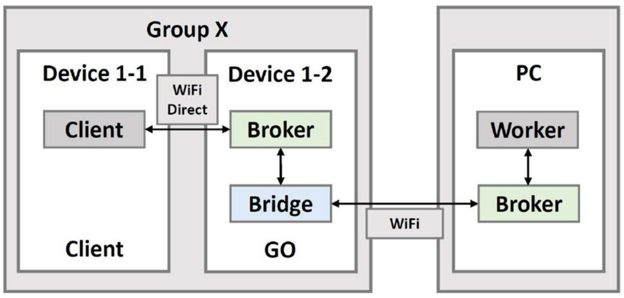


Fig. 8 A simple way to bridge device to PC

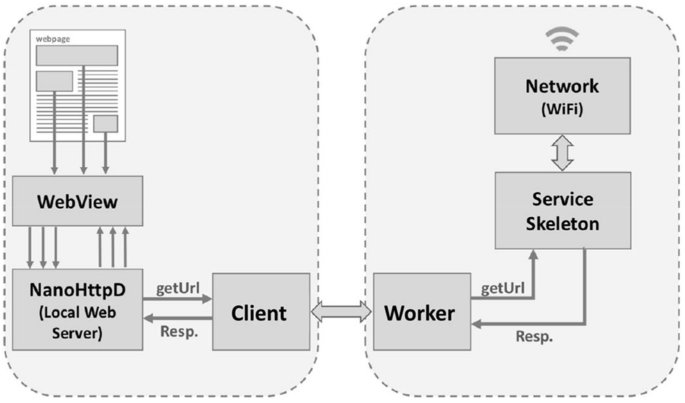


Fig. 9 Design of the remote browser app

sub links from the page contents. For each `getUrl` function call, the local server creates a request, distribute it to nearby peers, then collects and merges the results in reverse back to the `WebView` (Fig. 9), in the same way that a browser loads a web page locally.

Chat App

The Chat App simply sends and receives messages between nearby devices without any Internet connections. Figure 10 shows the architecural overview of the Chat App. The application wraps each message by a `UserMessage` object, each containing a message string, user info including `userId` and `username`, and a function named `createAt(time)`. Finally, we designed the `sendMessage` method as follows

- The Client sends a new message to Worker with a timestamp.
- The Worker first stores the new message inside a `Message Circular Buffer`.
- Then Worker searches for all the messages that newer than the `recvTime`. This searching process takes `O(n)` time because the newest messages are always at the front of the buffer.
- The `sendMessage` method returns a list of the latest messages.
- The Client loads the received list of the latest messages.

```
public UserMessage[] sendMessage(UserMessage msg, long recvTime) {
  if (messageList == null) {
    messageList = new ArrayList<>();
  }
  // insert the new message to Message List
  if (!msg.message.equals(EMPTY)) {
    if (messageList.size() >= MESSAGELIST_MAX_SIZE) {
      messageList.remove(0);
    }
    messageList.add(msg);
  }
  // get the newest messages
  List<UserMessage> cMsgList = new ArrayList<>();
  for (int i = messageList.size() - 1; i >= 0; i--) {
    UserMessage cMsg = messageList.get(i);
    if (cMsg.createAt > recvTime) {
      cMsgList.add(0, cMsg);
    } else {
      break;
    }
  }
  return cMsgList.toArray(new UserMessage[] {});
}
```

Code Snippet 7: Sample code for `sendMessage` function.

Figure 11 shows the final results of the two applications

Evaluation

For the evaluation, we built a testbed with Wi-Fi Direct featured by Android devices to evaluate the performance of the developed middleware system. A personal computer is also included to examine the bridge between mobile devices and stationary computers (Table 1).

Micro benchmarks

We designed a simple service with one function accepting a binary array as an input parameter and returning the size of the array. When forwarding a function call, a component (e.g. Client) packs and send the function message with parameter values out to another one. For this benchmark, we gradually increased the size of the binary array from 1KB to 1MB in order to figure out the network performance of the components. The measured time $T_{[Total]}$ will be estimated at the Client following the Eq. 1, with $T_{[Net]}$ being the total network round-trip time of all components.

$$T_{[Total]} = T_{[Broker]} + T_{[Bridge]} + T_{[Worker]} + T_{[Net]} \quad (1)$$

For the overhead measurement of each component, we isolated the network usage by running all components on a single device. Fig. 12-1 shows the promising result in which the Broker only spends 5 to 30 ms to store and forward a request while the Client and Worker steadily increase the processing time as the message size dilates over time, 18 to

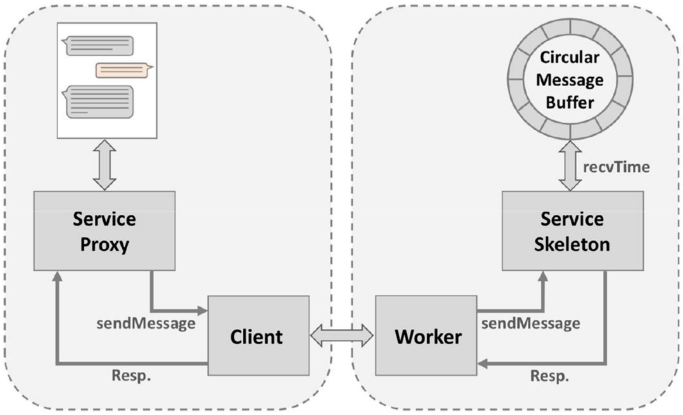


Fig. 10 Architecture design of the Chat App

240 ms and 5 to 90 ms respectively. When more devices join the collaboration, the message dispatching over the network significantly degrades the performance from 10 to 15 times slower (Fig. 12-2).

Devices to PC

An arbitrary device in a group may by chance be connectable with a stationary server, which enriches the group with more powerful resources. To make the server available, one Broker will be installed there along with the Worker(s) to receive requests and forward responses as in Fig. 8. The device contacting server will hold one Broker and a Bridge to forward requests from its Broker to the server's Broker.

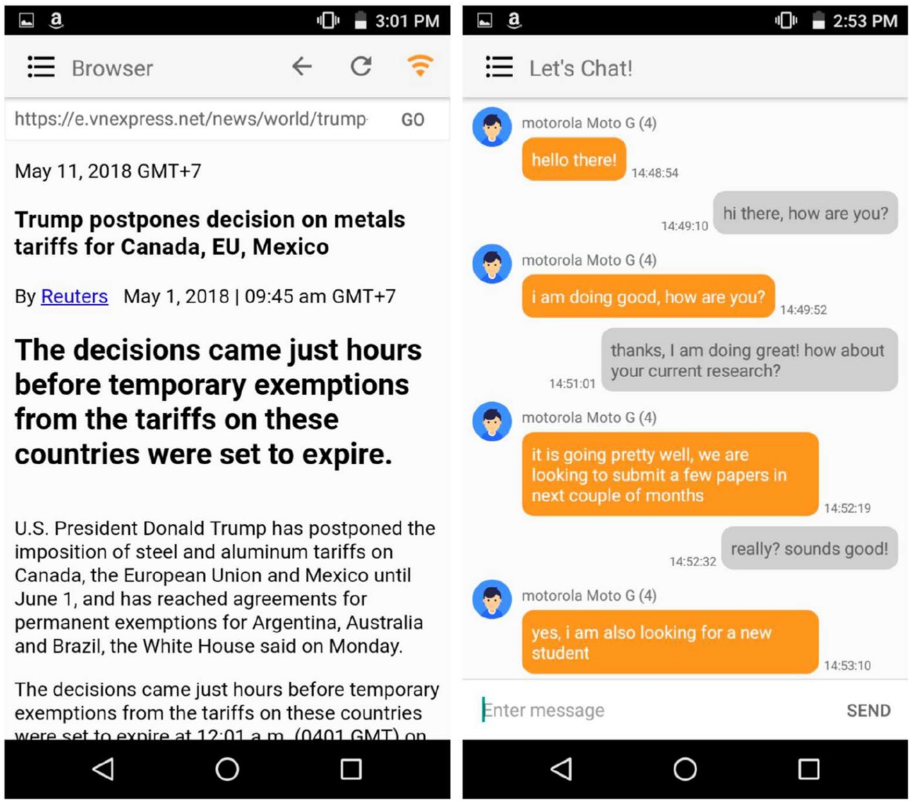
We, then compare the speed of Device(s)-to-PC over Wi-Fi with D2D over WiFi-Direct in the same network, Fig. 13 shows two cases: (1) the performance of one device to a PC is always better by from 2.2 to 4.5 times depending on message sizes; likewise, (2) two devices to a PC also performs 1.9 to 2.7 times faster.

Our middleware vs. RMI

Because the Android platform has limited APIs and does not support RMI, we proceeded the comparison between our middleware system and RMI on a typical server environment in which two servers periodically execute remote procedure calls on each platform. This experiment relies on the $T_{[Total]}$ value measured on the Client for four different tasks: (1) sending messages with *empty function* returning only the message size and gradually-increasing message sizes, (2) blurring an image, (3) detecting motions in two images using OpenCV[2] and (4) counting the most frequent words in a document.

Figure 14 depicts the differences between the two technologies. In the first test case, since the empty function returns the result immediately, the $T_{[Total]}$ is accumulated by mostly the network time and system overhead. In general, Java RMI has the less overhead which performs 70.8% faster than our middleware, but these overheads are

[2] OpenCV for Java: http://opencv-java-tutorials.readthedocs.io.



https://e.vnexpress.net/news/world/trump

May 11, 2018 GMT+7

Trump postpones decision on metals tariffs for Canada, EU, Mexico

By Reuters May 1, 2018 | 09:45 am GMT+7

The decisions came just hours before temporary exemptions from the tariffs on these countries were set to expire.

U.S. President Donald Trump has postponed the imposition of steel and aluminum tariffs on Canada, the European Union and Mexico until June 1, and has reached agreements for permanent exemptions for Argentina, Australia and Brazil, the White House said on Monday.

The decisions came just hours before temporary exemptions from the tariffs on these countries were set to expire at 12:01 a.m. (0401 GMT) on

motorola Moto G (4)
hello there! 14:48:54

14:49:10 hi there, how are you?

motorola Moto G (4)
i am doing good, how are you? 14:49:52

14:51:01 thanks, I am doing great! how about your current research?

motorola Moto G (4)
it is going pretty well, we are looking to submit a few papers in next couple of months 14:52:19

14:52:32 really? sounds good!

motorola Moto G (4)
yes, i am also looking for a new student 14:53:10

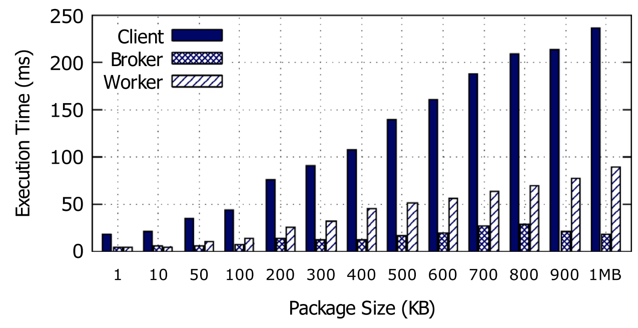
Fig. 11 The final results of the two Apps

Table 1 Specifications of the testing devices

	CPU	RAM	Battery
LG Volt	Quad-core 1.2 GHz	1 GB	3000 mAh
ZTE Maven 3	Quad-core 1.1 GHz	1 GB	2115 mAh
Moto G4	Octa-core 1.5 GHz	2 GB	3000 mAh
BLU R1	Quad-core 1.3 GHz	2 GB	2500 mAh
Dell PC	Intel i7-4790 3.6 GHz	8 GB	Wall-plugged

neglectful because it takes 10 to 35 ms by our middleware system and 3 to 10 ms by RMI for message sizes from 1 K to 1 MB. In the next experiment, we tested with two image processing test cases. Because the image processing task takes approximately 100 to 200 ms, the impact of overheads becomes trivial. The results of forty attempts (Fig. 14-2, 3) show slightly better performance of RMI compared with our middleware: 7.4% better for the image blurring and 11.2% for the motion detection.

Regarding the word counting service, the average time to examine each 1 MB document takes 2000 to 2500 ms which literally makes them futile. The results of forty attempts of word counting show the RMI is 0.4% faster than our middleware system, and thus the overall performance of our system is comparable with the Java RMI.



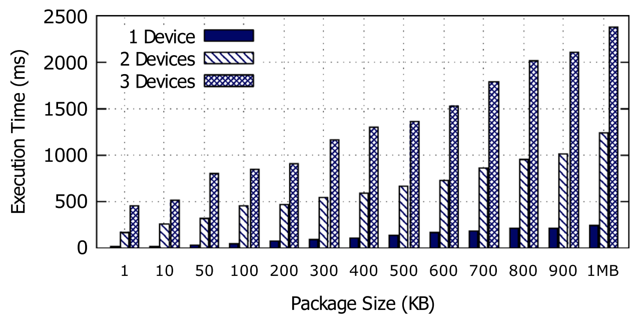
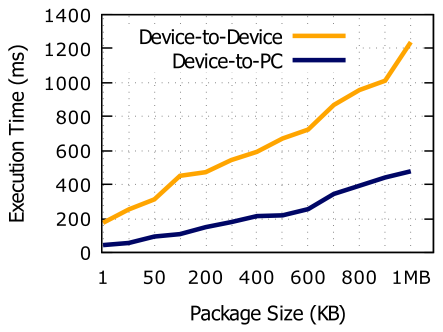


Fig. 12 Benchmarks: (1) run all components on one device. (2) Performance with multiple devices



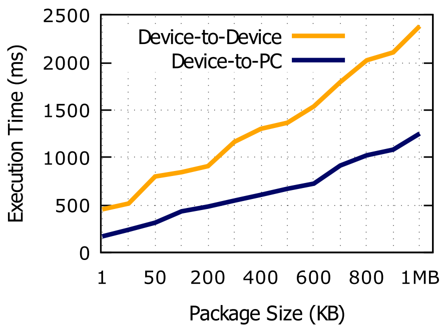


Fig. 13 Benchmarks: (1) run all components on one device. (2) Performance with multiple devices

Conclusion

In this article, we introduced a new middleware architecture to enable routing remote method invocation over multiple group device-to-device networks. Our work modularizes its architecture by the functional components using annotations, which makes it flexible to apply and adaptive with either D2D or device-to-server networks. Our case-study applications and empirical experiments tested on the actual testbed and real mobile devices unveil the ease of using the library in mobile software development, the low overhead conduction and high performance. In the future, we will consider these following directions (1) optimize the selection algorithm at Broker side to fairly distribute a request to a number of Workers for higher availability of mobile execution resources and better performance. (2) Secondly, to aim for large-scaled IoT networks, the middleware must comply with multiple criteria and test scenarios from a huge number of devices and different specifications, as well as rapid changes of device location and environment.

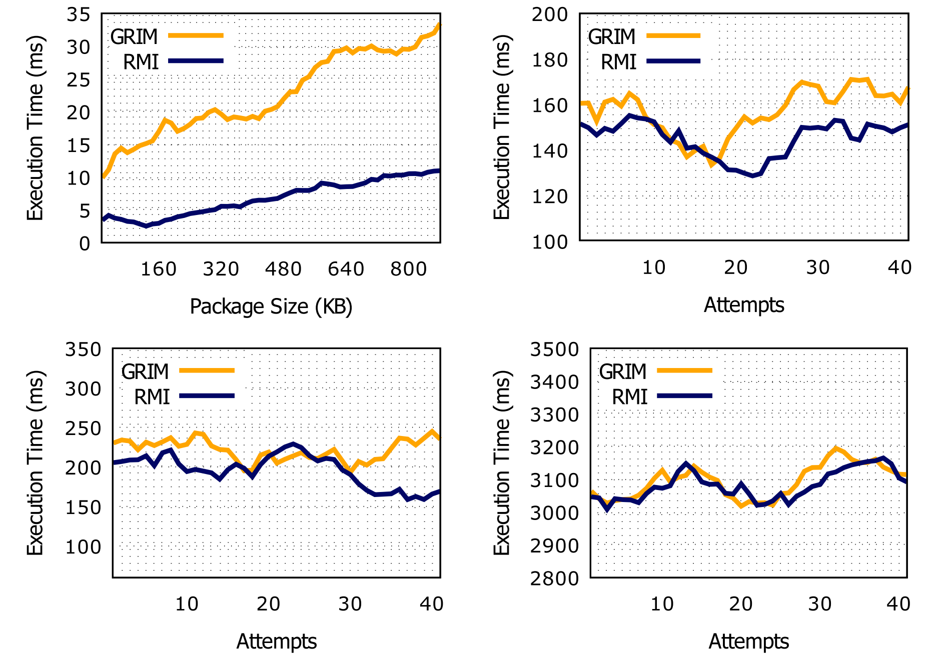


Fig. 14 Our middleware vs. `RMI`: (1) sending simple packages, (2) image blurring (3) motion detection, (4) word counting

This requirement can be resolved by a simulation where device specs and mobility are simulated in a predefined environment with several prerequisite conditions. Finally, (3) we will extend middleware library with respect to streamlining and easiness to target a larger scope of applications, by simplifying the APIs to reduce the cost of integration and development.

Authors' contributions

ML wrote the initial manuscript and developed the main system. SC co-wrote the manuscript and helped design the system and Y-WK is the principal investigator of the project and revised the manuscript. All authors read and approved the final manuscript.

Funding

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) grant funded by the Ministry of Education (NRF-2017R1C1B5075658).

Availability

The dataset and source code used in this research are available from the first author on reasonable request.

Competing interests

The authors declare that they have no competing interests.

Author details

[1] Department of Computer Science, Utah State University, Logan, UT, USA. [2] Department of Computer Science and Engineering, Kyungpook National University, Daegu, South Korea.

Received: 24 December 2018 Accepted: 6 May 2019
Published online: 05 June 2019

References

1. 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: MobiSys, New York, NY, USA, pp 49–62

2. Gordon MS, Jamshidi DA, Mahlke S, Mao ZM, Chen X (2012) COMET: code offload by migrating execution transparently. In: 10th USENIX symposium on operating systems design and implementation (OSDI), Hollywood, CA, pp 93–106
3. Calice G, Mtibaa A, Beraldi R, Alnuweiri H (2015) Mobile-to-mobile opportunistic task splitting and offloading. In: 11th WiMob, pp 565–572
4. Le M, Kwon YW (2017) Utilizing nearby computing resources for resource-limited mobile devices. In: Proceedings of the symposium on applied computing (SAC), Marrakech, Morocco, pp 572–575
5. Shi C, Habak K, Pandurangan P, Ammar M, Naik M, Zegura E (2014) COSMOS: computation offloading as a service for mobile devices. In: Proceedings of the 15th ACM international symposium on mobile ad hoc networking and computing 2014, Philadelphia, USA
6. Baresi L, Derakhshan N, Guinea S (2016) WiDiSi: a wi-fi direct simulator. In: IEEE wireless communications and networking conference, pp 1–7
7. Vinoski S (1997) CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Commun Mag 35(2):46–55
8. Baresi L, Derakhshan N, Guinea S, Arenella F (2017) Mag-net: a middleware for the proximal interaction of devices based on wi-fi direct. In: IEEE international conference on communications, pp 1–7
9. Rostanski M, Grochla K, Seman A (2014) Evaluation of highlyavailable and fault-tolerant middleware clustered architecturesusing RabbitMQ. In: 2014 federated conference on computer science and information systems, pp 879–884
10. Henjes R, Schlosser D, Menth M, Himmler V (2007) Throughput performance of the ActiveMQ JMS server. Kommunikation in Verteilten Systemen (KiVS). Springer, Berlin, pp 113–124
11. Kim YG, Kim DH, Lee EK (2017) Designing test methods for IT-enabled energy storage system to evaluate energy dynamics. J Inf Process Syst 13(6):1487–1495
12. Kaur J, Kaur K (2017) A fuzzy approach for an IoT-based automated employee performance appraisal. CMC Comput Mater Continua 53(1):23–36
13. Vanus J, Belesova J, Martinek R, Nedoma J, Fajkus M, Bilik P, Zidek J (2017) Monitoring of the daily living activities in smart home care. Hum Centric Comput Inf Sci 7(1):30
14. Funai C, Tapparello C, Heinzelman W (2016) Supporting multi-hop device-to-device networks through wifi direct multi-group networking. arXiv:1601.00028
15. Le M, Song M, Kwon Y (2017) Enabling flexible and efficient remote execution in opportunistic networks through message-oriented middleware. In: 2017 IEEE 41st annual computer software and applications conference (COMPSAC), pp 979–984
16. Le M, Song Z, Kwon Y, Tilevich E (2017) Reliable and efficient mobile edge computing in highly dynamic and volatile environments. In: Second international conference on fog and mobile edge computing (FMEC) 2017, pp 113–120
17. Casetti C, Chiasserini C, Pelle LC, Valle CD, Duan Y, Giaccone P Content-centric routing in wi-fi direct multi-group networks. In: 2015 IEEE 16th international symposium on a world of wireless, mobile and multimedia networks (WoWMoM), pp 1–9
18. Phunchongharn P, Hossain E, Kim DI (2013) Resource allocation for device-to-device communications underlaying LTE-advanced networks. IEEE Wirel Commun 20(4):91–100
19. Boabang F, Nguyen H-H, Pham Q-V, Hwang W-J (2016) Network-assisted distributed fairness-aware interference coordination for device-to-device communication underlaid cellular networks. Mobile Inf Syst. 2017(2017):1821084. https://doi.org/10.1155/2017/1821084
20. Kang HE, Jeong K, Lee K, Park S, Kim Y (2016) Android RMI: a user-level remote method invocation mechanism between android devices. J Supercomput 72(7):2471–2487
21. Lin TY, Chen J, Liu JH (2016) Enabling cooperative computing for android-based mobile platforms. In: 2016 international symposium on computer, consumer and control (IS3C), pp 763–766
22. Kim HW, Jeong YS (2018) Secure authentication-management human-centric scheme for trusting personal resource information on mobile cloud computing with blockchain. Hum Centric Comput Inf Sci 8(1):1–12
23. Nakao K, Nakamoto Y (2012) Toward remote service invocation in android. In: 2012 9th international conference on ubiquitous intelligence and computing and 9th international conference on autonomic and trusted computing, pp 612–617
24. Nagahara Y, Oyama H, Azumi T, Nishio N (2013) Distributed intent: android framework for networked devices operation. In: 2013 IEEE 16th international conference on computational science and engineering, pp 651–658
25. Choi J, Park J (2013) A framework for remote service invocation of android services to communicate with external services in java environment. J Inf Technol Serv 12(2):349–359
26. Toyama M, Kurumatani S, Heo J, Terada K, Chen EY (2011) Android ASA server platform. In: 2011 IEEE consumer communications and networking conference, pp 1181–1185
27. Teófilo A, Remédios D, Paulino H, Lourenço J (2015) Group-to-group bidirectional wi-fi direct communication with two relay nodes. In: MOBIQUITOUS 2015, Coimbra, Portugal, pp 275–276
28. Jeong M, Ahn S (2017) A network coding-aware routing mechanism for time-sensitive data delivery in multi-hop wireless networks. J Inf Process Syst 13(6):1544–1553
29. McNamara L, Mascolo C, Capra L (2008) Media sharing based on colocation prediction in urban transport. In: Proceedings of the 14th ACM international conference on mobile computing and networking, MobiCom '08, San Francisco, California, USA, pp 58–69
30. Yu CH, Doppler K, Ribeiro CB, Tirkkonen O (2011) Resource sharing optimization for device-to-device communication underlaying cellular networks. IEEE Trans Wirel Commun 10:2752–2763 8 pages
31. Cuomo F, Martello C, Baiocchi A, Capriotti F (2006) Radio resource sharing for ad hoc networking with UWB. IEEE J Sel Areas Commun 20(9):1722–1732 11 pages
32. Amiri Sani A, Boos K, Yun MH, 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 (MobiSys), New Hampshire, pp 259–272

33. Zhang N, Lee Y, Radhakrishnan M, Balan RK (2015) GameOn: P2P gaming on public transport. In: Proceedings of the 13th annual international conference on mobile systems, applications, and services (MobiSys), Florence, Italy, pp 105–119
34. Khan AJ, Jayarajah K, Han D, Misra A, Balan R, Seshan S (2013) CAMEO: a middleware for mobile advertisement delivery. In: Proceeding of the 11th annual international conference on mobile systems, applications, and services (MobiSys), Taipei, Taiwan, pp 125–138
35. Simoens P, Xiao Y, Pillai P, Chen Z, Ha K, Satyanarayanan M (2013) Scalable crowd-sourcing of video from mobile devices. In: Proceeding of the 11th annual international conference on mobile systems, applications, and services (MobiSys), Taipei, Taiwan, pp 139–152
36. Montresor A, Jelasity M (2009) Peersim: a scalable p2p simulator. In: IEEE nineth international conference on peer-to-peer computing 2009, pp 99–100
37. Hintjens P (2013) ZeroMQ: messaging for many applications. O'Reilly Media Inc., Newton

Publisher's Note