Distributed Computation in Mobile Platforms

Kaiyuan Sun

April 2023

1 Introduction

Mobile devices, such as smartphones and tablets, have become an essential part of our daily lives. With the growing demand for faster and more powerful mobile applications, distributed computation has become an increasingly popular technique for developing mobile apps that require intensive computation.

Distributed computation in mobile platforms enables the offloading of computation tasks from mobile devices to remote servers or other connected devices, allowing for faster processing times, reduced power consumption, and improved user experience. This technology has opened up new possibilities for mobile app development, including advanced data analytics, machine learning, and artificial intelligence applications.

Compared to traditional servers, difficulties in mobile platforms include a) unpredictable periods of unavailability because a mobile device will power off or lose network connection more frequently; b) higher cost of wireless transportation; c) volatile data because a mobile device can join or leave the system at anytime; d) harder to get support from different hardwares and platforms, compared to Linux servers.

Despite all these disadvantages, it is still necessary to perform distributed computation in mobile platforms. In this paper, we will first introduce some distributed computation and concurrent computation models, and then explore the use of these models in mobile platforms.

2 Distributed concurrent computation

2.1 Actor Model

Actor Model[1] is one of the earliest and most influential model for distributed computation, and has recently gained popularity again because of the success achieved by Erlang, its flagship programming language. Actor model was first proposed in 1980s. The basic idea is "Everything is an actor": an application, a device or an object in a program can be an actor; the model provides stronger promises about concurrent computation node such as data race freedom when

compared to traditional shared-memory-based abstractions like the fork-join model.

In Actor Model, an actor can perform the following actions:

- Send finite messages to other actors
- Create finite new actors
- Designate what to do with the next message

The Actor Model decouples messages and the nodes that send messages by encapsulating mutable state into actors and using asynchronous messaging to coordinate activities. The sending end dispatches the tasks to actors and then gathers the messages containing computation results. Some people believe that this is what inspires the Message Passing Interface(MPI).

2.2 Selector

The Actor Model is a lock-free model, which means it lacks guarantees to control the order in which messages are processed by an actor, thus making it difficult to implement synchronization and coordination patterns. Selectors[5] are extension of actors, with multiple mailboxes and each mailbox guarded. With selectors, it is much easier to deal with synchronization and coordination patterns, such as the typical producer-consumer model.

Selectors are the extension of actors. The biggest difference between a selector and an actor is the mailbox. A selector has multiple mailboxes where messages are sent to. Sending messages to any of the mailboxes is a non-blocking operation, so that the messages can be received concurrently. A mailbox is guarded by a boolean guard. This guard can enable or disable a mailbox. When a mailbox is disabled, the selector cannot consume a message from it, so the selectors only process messages from active mailboxes. By this definition, a standard actor can be viewed as a selector with one single mailbox.

The specific mailbox where a message goes to can be decided either by the sender, if specified as an argument, or by the recipient selector if not. Sending a message to a disabled mailbox will block. By this mechanism, programmers can implement synchronization easily. For example, when dealing with producer-consumer problem, we can define the producer and consumer as two actors, and the buffer as a selector with two mailboxes. These two mailboxes accept messages from the producer and consumer respectively. When the buffer is empty, the mailbox of the consumer is disabled, while when the buffer is full, the mailbox of the producer is disabled. After each process of a message in the buffer, we update both guards.

Another example will be Request-Response pattern. With traditional Actor Model, if we want synchronous request-response, the sender must stall after it sends a request until it receives the response. This pattern is very inefficient because the sender will not be able to receive messages from other actors when waiting for the response. But with selectors, we can handle this elegantly.

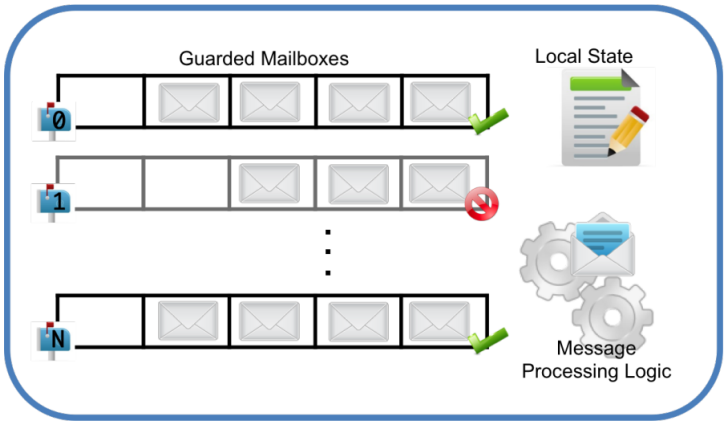

Figure 1: Selector Model

The sender, as a selector, can have two mailboxes, one to receive regular messages and the other one to receive synchronous response messages. When it is expecting a response, it simply disables the regular mailbox to ensure the synchronization. The response will go to the response-mailbox which is specified by the receiving end of the request. After the response is processed, the regular mailbox can be enabled again.

The Actor Model and its selector extension have become the inspiration of many distributed concurrent computation models and frame works.

3 Practice of Distributed Computation in Mobile Platforms

There are many limitations in mobile devices, such as power, network connection and authority settings, which makes mobile devices difficult to handle efficient computation tasks. However, in some realms, specific tasks can be performed in mobile platforms, like P2P file transportation[7] and the off-the-grid agriculture devices[8]. There are also many studies to explore a more general and scalable distributed strategy.

3.1 DAMMP

Distributed Actor Model for Mobile Platforms[4] (DAMMP) is the extension of Distributed Selector model in mobile platforms. In order to introduce DAMMP, we need to introduce HJ and DS first.

3.1.1 Habanero-Java

Habanero-Java[2] (HJ) is a dialect of Java, with corresponding compiler hjc and runtime environment hj runtime, which are all developed by Rice University. The HJ programming language aims to make up for the shortage of Java in parallel programming. Java provides only a low-level view of parallel programming, like threads, tasks, locks and atomic operations, and Java also provides no closures, making it onerous to write an elegant and simple parallel program. In the language level HJ proposes these following primitives:

- async ¡stmt¿: causes the parent task to create a new child task to execute ¡stmt¿ asynchronously with the remainder of the parent task
- future: a future variable f is a future handle to a newly created task by async. By calling f.get(), we will obtain the result of the task if it has completed, or block if not
- foreach/forall: provides parallel operation to all the values in a collection
- finish ¡stmt¿: codes after finish ¡stmt¿ will only be executed after all the asynchronous codes inside ¡stmt¿ are completed

To conclude, HJ is an extension to Java will stronger support for parallel programming.

3.1.2 Distributed Selector

The Distributed Selector model[3] is designed based on HJ runtime, to provide a scalable, reactive and versatile distributed computing with the proliferation of manycore/heterogeneous processors on portable devices and cloud computing clusters that can be elastically and dynamically allocated.

DS refers to each single HJ runtime instance as a place. A physical computing node can serve as one or multiple places, each place having its own address space and port.

Figure2 shows the basic structure of DS model. Every computing node is denoted as a place with many selectors, including System Actor, Proxy Actor and many other user-defined selectors.

The selector system, which refers to the HJ runtime instance on a single place, contains the System Actor and Proxy Actor. The System Actor maintains the internal state of the place's selector system and communicates such information to other selector systems. The Proxy Actor coordinates the messages between local and remote selectors. Each Proxy Actor records all the local selectors for easy communication, and communicates with a remote Proxy Actor to send messages to a remote selector. The remote Proxy Actor will then forward this message to its local selector instances.

There is a Master Node in this distributed selector system, which has a place responsible for global initialization and bootstrap, and for joining and leaving of nodes. When bootstrapping, the Master Node establishes SSH connection to

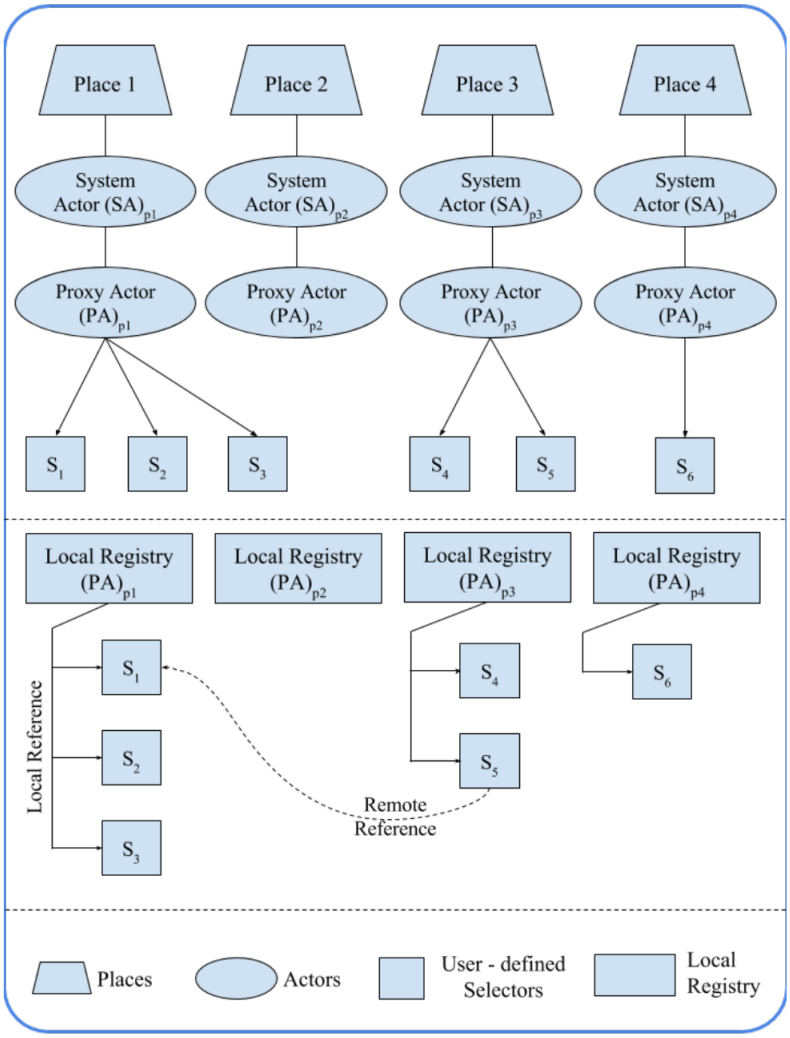

Figure 2: DS model

all the places it knows. A place will send an ack to master if it recognizes the Master. After the establishment of connection, the Master notifies the Proxy Actor of other places and tells it to start processing.

3.1.3 DAMMP

Distributed Actor Model for Mobile Platforms(DAMMP) is the extension of Distributed Selector model in mobile platforms. In DAMMP model, every mobile device is viewed as one single place, which is different from DS model, where a HJ runtime is a place, and a device can have multiple logical places. A device is composed of multiple user selectors, a System Actor, a Proxy Actor and a Mobile Communication Manager. The Mobile Communication Manager manages the communication between the local device and other remote devices.

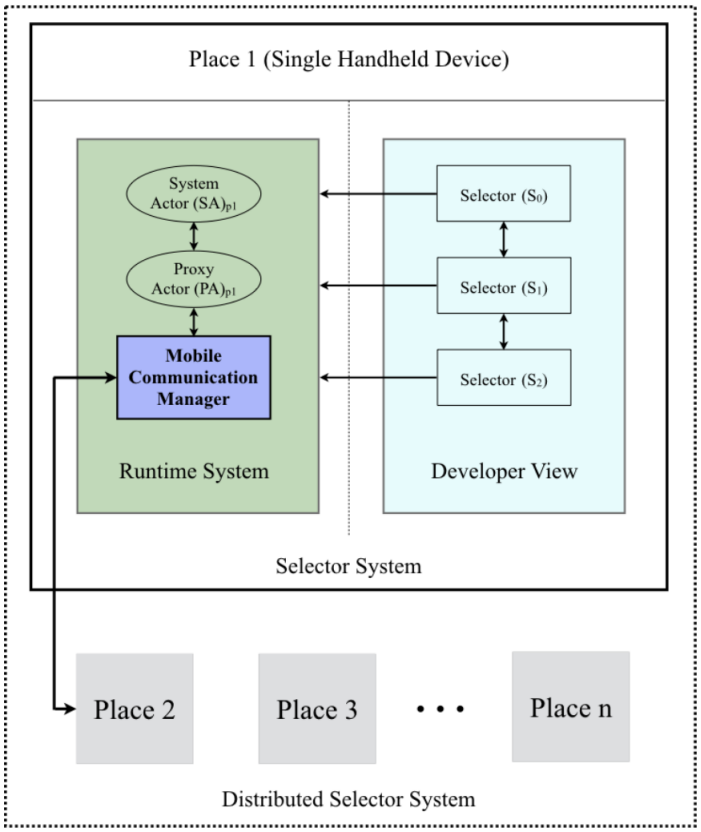

Figure 3: Overview of DAMMP

```
public interface IMobileCommunicationManager {
    interface ISystemCallback {
```

```
        void onConnectionReady ( Place localNode );
        void onMessage ( Message message );
        void onPlaceJoin ( Place place );
        void onPlaceLeft ( Place place );
    }
    void start ();
    void stop ();
    boolean send ( Place place , Message message );
    void setSystemCallback ( ISystemCallback callback );
}
```

There is a callback function which makes a place able to ignore the topological change in the network structure. It does not have to update the information immediately when there is a node joining or leaving the network; instead, it calls the callback function and updates the local network information only when there is a reference to the changed node.

DAMMP adopts a Ping-Pong model for message communication. The sending end is called Ping, and the receiving end is called Pong. As fig shows, Ping wants to create a "Pong selector" in the Pong device. They need their Proxy Actors to proxy the generation of selectors. Proxy Actors cannot communicate directly so they also need the help of Mobile Communication Manager.

There is a group owner in the system which stores the information of all the nodes inside the network. When a new node is ready to join, it first communicates with the owner and retrieves the information of other nodes, and then broadcast its existence by actively connecting other nodes.

The network structure is fine for computation. Now we need to decide which node to choose when we want to deliver a task to a computation node. This cannot be done by the programmer since the programmer doesn't know the information inside the network. DAMMP uses Mobile Communication Manager to solve this. The Mobile Communication Manager periodically broadcast the following messages of all nodes:

- Battery Status Message: Battery low / Battery okay
- Charging Status Message: Device is connected: AC or USB / Disconnected: AC or USB
- Battery Level Message: Battery percentage when battery level changes
- Temperature Message: Average temperature in last 's' seconds has changed by 'd' degrees in ∘C
- WiFi Signal Strength: Average WiFi signal strength (1 to 10) in last 's' seconds changed by 'w'

These information is able to decide whether a node is suitable for a given task.

Using the DAMMP runtime, application developers can tap into the processing power of other devices by creating a network of heterogeneous devices.

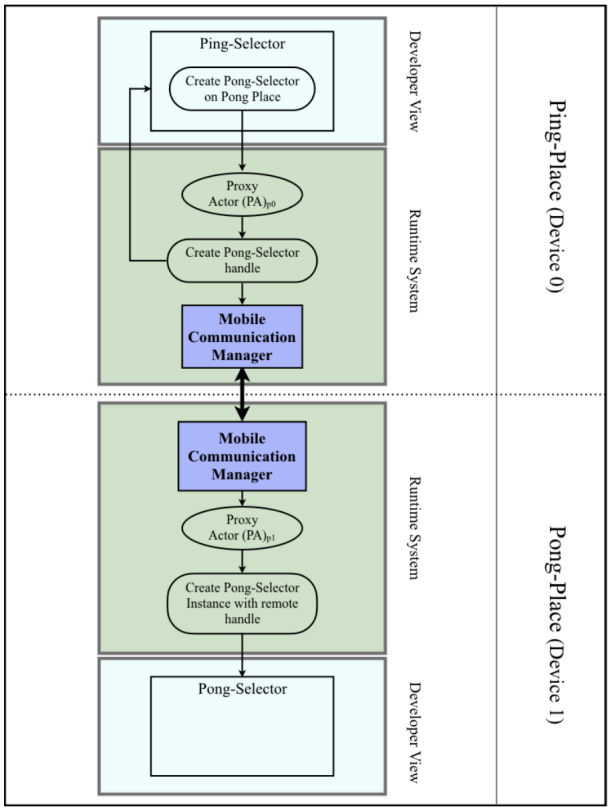

Figure 4: Ping-Pong Model

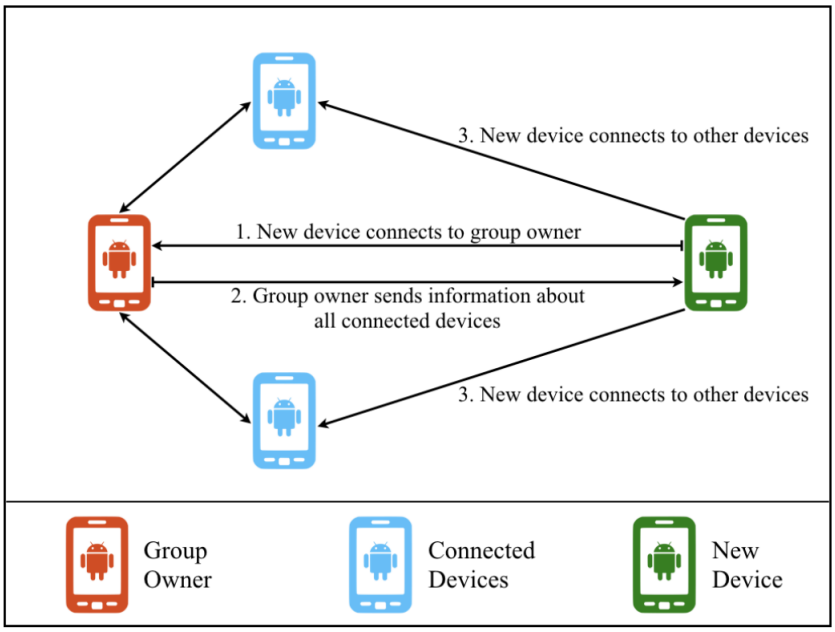

Figure 5: New device joins the network

Applications can readily harvest these computing resources to aid application demands that exceed the capability of on-device chips while also reducing battery consumption and addressing thermal constraints at the same time.

3.2 ActorNet

ActorNet[6] is an actor platform for wireless sensor networks (WSNs). A typical WSN may consist of hundreds to tens of thousands of tiny nodes embedded in an environment. There are restrictions to computation in WSNs, including a finite energy supply, low network bandwidth, and limited memory and processing power. The devices in a WSN are often cheap and cannot handle computation tasks. ActorNet provides a solution: a mobile agent platform. Agent programs migrate from node to node, and sample and process data on the spot. All data need not be collected centrally, and agents can be injected into the WSN on a need basis.

ActorNet provides higher level of abstraction to parallel and asynchronous programming, and is proved to have better performance than traditional stack-based virtual machines on sensors with limited power and memory.

4 Conclusion

Although there is no universal model for distributed computation in mobile platforms, many models are proposed to solve specific problems in specific areas.

There are still many problems for distributed computation in mobile platforms. For example, load balancing is more complicated than traditional distributed systems, where we only need to worry about the performance of servers; in mobile world, instead, we need to consider power, location, network, authority and many other things. Sometimes we may need a server to join the network, making the system even more complex.

However, with the growing demand for faster and more powerful mobile applications, there will definitely be more studies on this topic.

References

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT press, 1986.

[2] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61, 2011.

[3] Arghya Chatterjee, Branko Gvoka, Bing Xue, Zoran Budimlic, Shams Imam, and Vivek Sarkar. A distributed selectors runtime system for java applications. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 1–11, 2016.

[4] Arghya Chatterjee, Srdan Milaković, Bing Xue, Zoran Budimlić, and Vivek Sarkar. Dammp: A distributed actor model for mobile platforms. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, pages 48–59, 2017.

[5] Shams M Imam and Vivek Sarkar. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 1–14, 2014.

[6] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: An actor platform for wireless sensor networks. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1297–1300, 2006.

[7] Christoph Lindemann and Oliver P Waldhorst. A distributed search service for peer-to-peer file sharing in mobile applications. In *Proceedings. Second International Conference on Peer-to-Peer Computing,*, pages 73–80. IEEE, 2002.

[8] Richard K Lomotey, Yiding Chai, Ashik K Ahmed, and Ralph Deters. Distributed mobile application for crop farmers. In *Proceedings of the fifth international conference on management of emergent digital EcoSystems*, pages 135–139, 2013.