Report - Service composition in opportunistic networks

Author: Siddharth Kalluru
Advisor: Dr. Mohan Kumar

Introduction:

Problem statement:

Modern urban ecosystems are dominated by mobile devices like mobile phones, tablets, laptops etc. Each device, in terms of computation capability, is powerful enough to host few low level services that may be used by other mobile devices that may not have access to those services. Since these devices are mobile, they may not always be in contact with each other. However, when they come in contact with each other they can make use of services available with the devices in their vicinity.

Invoking services from neighbouring devices is certainly possible, but can we combine these services? Therefore the main goal of this project is to provide a framework for service composition in an opportunistic network.

Background:

In the recent years, service oriented computing has gained prominence over traditional component based architectures. The focus of component based architectures was to develop off the shelf components that could be easily reused across systems. However, these components, on their own served no purpose. A natural extension to this architectural pattern would be to develop functionality in the form of software components that had an individual purpose (i.e having the ability to be used on their own) and be reusable. This would allow for functionality to be composed to achieve newer functionality.

The number of mobile devices also has increased significantly in the last few years. Moreover, mobile phones these days are embedded with different sensors and communication technologies. For example, a standard android device has ambient light sensors, accelerometers for motion detection, bluetooth, Wi-Fi and 4G connectivity. The ubiquitous nature of these mobile devices has given rise to opportunistic networks. These networks are formed when devices communicate only when they have an opportunity for communication. In most cases, it’s when mobile devices come within transmission range of each other.

This project aims to extend the idea of composing services to the opportunistic networks that are formed by mobile devices.
Literature Review:

Kalasapur et al. [1] propose a mechanism to combine services in a pervasive environment using a graph based approach. Services are represented as edges and I/O params as nodes of a graphs. Their approach is to use these graphs to check if services can be combined together or not. They also categorized the various devices in an environment according to the features supported by them. A device with lower features or lower resources could piggyback on another device that has more resources. This approach was termed as ‘LATCH’ [1] process.

This approach of theirs to employ service composition using the ‘LATCH’ process is termed as ‘Hierarchical Service Composition’[1]. It is clear that the authors of this paper target a generic pervasive environment with heterogeneous devices with little emphasis on mobility of devices and the opportunistic networks created by them.

Tamhane et al.[2] designed a middleware targeted specifically at opportunistic networks. This middleware is completely modular and devices may choose to some or all the modules. The authors propose a handshaking mechanism on device contact. As part of this process the two devices exchange service graphs[2] with each other. This is essentially the exchange of information about the different services available on different devices irrespective of the presence of the device in the neighborhood.

The module to model the network as a graph[2] is also available. This allows as device employing the middleware by the authors to guess when the devices with services will be encountered again for invocation. Additionally this information can be used to generate a mobility pattern and provide a reasonable estimate of the probability of success for a given composition request. However, even though the authors tested the middleware using simulation and real-world testing of such middleware is still pending.

Sadiq et al.[3] focus on improving the efficiency of service composition in opportunistic networks by taking into account the load on the devices with services and temporal distance[3] between the devices. The concept of temporal distance[3] is focal point of the paper as it is used to select services that have a closer proximity.

Apart from this varying level service loads awareness is also considered. Support for encryption is also emphasized so as to ensure integrity of the data being passed around. The authors successfully demonstrated significant performance improvement over non load aware service composition in opportunistic networks[3]. However, the authors don’t provide a comprehensive design of the system wit the above capabilities.
Approach to service composition

I've used graph based approach can be used to identify the possible compositions amongst the services available. This approach is similar to the one proposed by Kalasapur[1] et.al.

A service graph[1] is a graph that has services represented as edges and the nodes being the input and output data. A composition is possible only a path exists in the service graph. Moreover, to increase the efficiency of the procedure to check for composable services, the entire service graph is divided into the following two levels. Therefore, for a composition to be possible a path (i.e a collection of edges or services) must exist in both levels of the service graph.

1. Semantic Level:
The nodes in this graph are represented by the semantic i/o parameters (eg. high level i/o description). For example consider a translation service that translates from the english to spanish. Here the input is in English language and the output is in Spanish language. At this level we’re only concerned about the meaning of input and the output. The semantic service graph for the above service can be seen in Figure 1.

![Figure 1](image1)

Similarly, given a list of translation services $<S_{Eng to Spa}, S_{Spa to Eng}, S_{Eng to Fr}, S_{Ger to Spa}>$ the semantic level of the service graph is represented as in Figure 2.

![Figure 2](image2)
2. Syntactic Level: The nodes in this level of the graph are represented by the syntactic i/o parameters (eg. i/o file specifications). At this level we’re concerned about the format of the input and the output of the services. For example, in the above list of services the input can be in text and the output can be in audio, in a different language. This can be seen in Figure 3.

Consider the example seen in Figure 2. There are 3 paths possible at the semantic level here. They are

1. \( S_{\text{Ger to Spa}}, S_{\text{Spa to Eng}}, S_{\text{Eng to Fr}} \rightarrow \) A composed service to translate from German to French
2. \( S_{\text{Ger to Spa}}, S_{\text{Spa to Eng}} \rightarrow \) A composed service to translate from German to English
3. \( S_{\text{Spa to Eng}}, S_{\text{Eng to Fr}} \rightarrow \) A composed service to translate from Spanish to French

Next step in checking for composition is to check the services for each of the above paths in the syntactic graph. If the German to Spanish translation service takes a text file as input and gives back an audio file in the spanish language then the syntactic graph for the above scenario is as follows.

It can be seen from Figure 4 that the composed translation from **German to French isn't possible** as there **isn't a path** from the German text node to the French text node in the **syntactic** graph. Same is the case for the composed translation of German to English.

However, since there exists a path, in the syntactic graph (Figure 4), from the node **Spanish text** to **French text**, a composed translation from Spanish to French is possible.

Extraction of paths from the graphs is a process of depth first search applied iteratively.
Pseudo code for the above approach is as follows.

//Service Definition

Service{
    String semIp, semOp;  //Semantic I/O parameters
    String synIp, synOp;  //Syntactic I/O parameters
    String serviceDesc   //Description of the service
}

//Node definition - to be used to construct graphs

Node{
    String nameOfIOParam;  //Node value
    Boolean visited;       //Flag to indicate if a node has been visited in the graph
}

//Pseudo code to create an adjacency list (Represented by a map) representing the semantic graph

getSemanticGraph (list <Service> services){
    semanticGraph <Node, list<Nodes>>  //Blank map of nodes to list of nodes
    For each (Service s in services){
        Node iNode = new Node(s.semIp);  //Create a node for the semantic input parameter
        Node oNode = new Node(s.semOp);  //Create a node for the semantic output parameter
        //Add the iNode as key in the semanticGraph map if it does not exists
        //Add the oNode as key in the map semantic graph map, if it does not exists
        //Add the oNode to the list of neighboring nodes for the iNode  ➔ indicates an edge for the current service
    }
    Return semanticGraph;
}
getSyntacticGraph (list <Service> services){
    syntacticGraph <Node, list<Nodes>> <Blank map of nodes to list of nodes>
    For each (Service s in services){
        Node iNode = new Node(s.synIp);      //Create a node for the syntactic input param
        Node oNode = new Node(s.synOp);      //Create a node for the syntactic output param
        //Add the iNode as key in the syntacticGraph map if it does not exists
        //Add the oNode as key in the map semantic graph map, if it does not exists
        //Add the oNode to the list of neighboring nodes for the iNode → indicates an edge for the current service
    }
    Return syntacticGraph;
}

getListOfSemanticCompositions(semanticGraph){
    List < List<Service>> listOfComposition;    //Create a blank list of compositions
    For each Node in semanticGraph.keys {      //keys → nodes in the semantic graph
        if(Node is not visited ){
            listOFServices = new List<Service>    //to accumulate services in the path
            //perform depth first search from current node N and accumulate the services encountered while recursively performing depth first search
            //Add the accumulated list of services into the list of compositions
        }
    }
    Return listOfCompositions
Main method

Services = array(Service); //Given a list of services

semanticGraph = getSemanticGraph(Services) //Construct the semantic graph → a map

of

// node to the list of adjacent nodes

syntacticGraph = getSyntacticGraph(Services)

listOfSemCompos< list< Service> > = get不但SemanticCompositions(semanticGraph)

For each list<Service> semServices in listOfSemCompos{

- Check in the syntacticGraph if the path between start and end service exists or not
  using Depth first search, beginning with the first Service S in the list semServices.
- If the path does not exist in the syntacticGraph then remove it from the list of compositions.

}

Print (listOfSemCompos) //the list here contains the compositions that have been filtered according to the syntactic graph
**Proposed solution and Implementation details:**

**Design:**

This section discusses the high level design of the proposed solution for service composition in opportunistic networks.

The prototype was developed as an Android application, but the approach is not limited to this Android implementation.
The description of the components are as follows.

1. **User Interface**: User interface module is the collection of components that are responsible for interaction with a user. With respect to Android, it’s the collection of all files related to Activities (.java and .xml files)

2. **Service Composition Handler**: Generally an object that maintains the state or context of the entire service composition application. This has been implemented as a singleton plain old java object and its reference is made available to any module that needs it.
   a. **Service graph**: This module is responsible for the application’s awareness of all the services available to it. A list of individual service objects is maintained. Each service object contains details such as service description, semantic i/o parameters, syntactic i/o parameters, location, cost of service and delay associated with it. From the list of service objects, graphs based on semantic and syntactic i/o parameters are calculated. Finally, using iterative depth first search, a list of all possible service compositions is obtained and displayed to the user. Both individual services and service graphs are represented as plain java objects.
   b. **Request queue**: This queue is responsible for holding all the pending service requests issued by a device. A concurrent hash-map was used for this purpose. The key is control message corresponding to the request and the value is the name of the associated file.
   c. **Response queue**: This queue is responsible for holding all the pending responses issued by a device. A concurrent hash-map is used for this purpose. The key is control message corresponding to the request and the value is the name of the associated file.
   d. **Proxy queue**: This queue is responsible for holding the data that needs to forwarded along to the next node. A concurrent hash-map was used for this purpose. The key is control message corresponding to the request and the value is the name of the associated file.
   e. **Acknowledgement queue**: This queue is responsible for holding all the acknowledgements for various requests issued by a device. A concurrent hash-map was used for this purpose. The key is control message corresponding to the request and the value is the time-stamp at which the acknowledgement was issued.

3. **Concurrent task performers**: Tasks that need to be performed concurrently fall in this category. This allows the user interface to be decoupled with the networking aspects of the application. All of the below task performers are represented as threads that come into existence when the application is started and continuously run in the background.
   a. **Service graph collector**: This thread is responsible for periodically collecting the service graphs from the neighbouring mobile devices. With
respect to the Android prototype, this thread checks if any of the paired mobile devices are available in the neighborhood and exchanges service graph with them.

b. **Service invoker:** This thread is responsible for flushing out data from ‘Request Queue’. With respect to the Android prototype, this thread falls asleep for 3 seconds if the queue is empty, else tries to forward the files to the appropriate devices in the neighborhood for service invocation. This task is repeatedly performed.

c. **Response dispatcher:** This thread is responsible for flushing out the ‘Response Queue’. With respect to the Android prototype, this thread falls asleep for 5 seconds if the ‘Response Queue’ is empty, else tries to forward the files to the appropriate devices in the neighborhood for service invocation. This task is repeatedly performed.

d. **Message forwarder:** This thread is responsible for flushing out the ‘Response Queue’. With respect to the Android prototype, this thread falls asleep for 7 seconds if the Proxy Queue’ is empty, else tries to forward the files to the appropriate devices in the neighborhood for service invocation. This task is repeatedly performed.

e. **Acknowledgement Dispatcher:** This thread is responsible for flushing out the ‘Acknowledgement Queue’. With respect to the Android prototype, this thread falls asleep for 10 seconds if the ‘Acknowledgement Queue’ is empty, else tries to forward the files to the appropriate devices in the neighborhood for service invocation. This task is repeatedly performed.

All Concurrent helpers make sure to not write duplicate messages into their respective queues.

4. **Helpers:**
   a. **Bluetooth helper:** This module deals with all the networking functionality. With respect to the Android prototype, this java object has methods to communicate with other mobile devices over bluetooth. This includes transfer of control messages and data between mobile devices.

   b. **File I/O helper:** This module deals with all the local file input and output operations. It is represented as a java object and exposes method to read and write files to the external storage in the Android prototype.

5. **Incoming request handler:** This thread is the most important handler as it responsible for the incoming connections. The logic for checking the incoming control message and the decision that has to be taken based on the control message is present here.
   a. It evaluates the control message to check if the intended device has received the request then the request is serviced and response is written out to the ‘Response queue’.

   b. Similarly, if the destination is not reached and a request is encountered, it is written to the ‘Proxy queue’ for forwarding.

   c. If an acknowledgement is received, then control messages corresponding to the acknowledgement are removed from all queues and then adds the current acknowledgement to the acknowledgement queue.
Protocol:

All communication between mobile devices running the prototype application happens in 2 parts. The first part is the control message and the second part is the file containing the data related to the control message. The format of the control message is discussed in detail in the next section.

Message format:

The structure of the control message in the first part of the communication is as follows.

<table>
<thead>
<tr>
<th>Control message code</th>
<th>Request ID</th>
<th>Bluetooth Source Address</th>
<th>Bluetooth Destination Address</th>
<th>Size of incoming file</th>
<th>List of Service objects</th>
<th>Request Timestamp</th>
<th>Max Number of hops</th>
</tr>
</thead>
</table>

1. **Control message code**: This field takes one of the values from the set <HANDSHAKE, REQ, RESP, ACK>. The value HANDSHAKE lets the device know that a device wants to perform a handshake by exchanging their service graphs. Similarly, the codes REQ, RESP and ACK indicate service requests, responses and acknowledgements.

2. **Request ID**: This field contains the unique request identifier and is used to drop the ignore the incoming messages that are duplicate.

3. **Bluetooth source address**: This field holds the address of the node that is sending the control message. With respect to the prototype, this is field contains the string representation of the bluetooth address.

4. **Bluetooth destination address**: This field holds the address of the node that is recipient of the control message. With respect to the prototype, this is field contains the string representation of the bluetooth address.

5. **Size of incoming file**: This field indicates the size of the file that will follow the current control message.

6. **List of Services object**: This field contains an ordered list of services that in the order the composition is to be performed. The first service in the list indicates the first service that needs to be performed on the accompanying data file. Similarly, the second third and other services represent the order in which the accompanying data is to be transformed.

7. **Request timestamp**: The unix timestamp as a long value is embedded in this field.
8. **Max number of hops**: This field specifies the max number of hops that a message is allowed to take. At every hop this field is decremented by 1. If this field reaches 0 then the message is dropped even if it's encountered for first time.

**Example Scenario**

For example, Device B has a service to translate text from English to Spanish and Device C has a service to translate from Spanish to German.

On the second handshake as seen below, Device A computes that a composed service for translating text from English to German is available. Device A then issues a request for such a service and sends the **english text in a file after the initial request control message**.

Device B performs its **intermediate conversion** and forwards it to Device C *(when it becomes available)* for the final conversion. Device B removes its service from the **list of services** in the control message to indicate that it has finished with its transformation on the underlying data. Every-time a transformation is performed on the underlying data, the list of services in the control message is updated and then the control message is forwarded along.

Finally, when Device C becomes available, it performs the **final transformation** on the accompanying data in the file. It then converts the **request control message** to **response control message** and forwards it along.

No transformations are performed on the response control messages and are just forwarded along till they reach the source address. In our case, the response control message goes through Device B and arrives at Device A.

When the response arrives at Device A, it issues an acknowledgement for the request that it'd created and forwards it along. On receiving an acknowledgements, the intermediates purge all their queues of messages corresponding to the request id mention in the acknowledgement control message.

A sequence diagram showing the interaction between devices is shown below. Dotted vertical lines represent the passage of time and the solid rectangular boxes on the dotted ones indicate the time when the devices were active.
Testing:

Given enough time all composition request would succeed. However, for any sort of real world application the compositions under test are evaluated with a **time limit of 2 minutes**. Also, it was set up such that only mobile device 2 would come in contact with 1 or 3 and **mobile device 1 and 3 would not come in contact** with each other. Hence for some compositions, mobile device 2 would be working as a **proxy** for the other two devices.

The framework was **tested for robustness** by knocking off one of the nodes during composition. This was done to simulate nodes moving **out of communication range**.

<table>
<thead>
<tr>
<th>Mobile Device #</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile Device 1</td>
<td>1. Spanish to German (.txt to .txt)</td>
</tr>
<tr>
<td></td>
<td>2. Portuguese to Czech (.rtf to .rtf)</td>
</tr>
<tr>
<td></td>
<td>3. French to Italian (.txt to .txt)</td>
</tr>
<tr>
<td>Mobile Device 2</td>
<td>1. German to Italian (.txt to .rtf)</td>
</tr>
<tr>
<td></td>
<td>2. German to Portuguese (.txt to .rtf)</td>
</tr>
<tr>
<td></td>
<td>3. Finnish to French (.txt to .txt)</td>
</tr>
<tr>
<td>Mobile Device 3</td>
<td>1. Italian to Finnish (.rtf to .txt)</td>
</tr>
<tr>
<td></td>
<td>2. French to English (.txt to .txt)</td>
</tr>
<tr>
<td></td>
<td>3. Czech to Italian (.rtf to .rtf)</td>
</tr>
</tbody>
</table>

Based on the **services from all 3 devices**, the semantic service graph for the above services is as follows.

![Figure 7](image_url)
Corresponding syntactic graph is as follows. The file formats are mentioned within parenthesis inside each node of the graph.

![Diagram of the syntactic graph](image)

Compositions from the above services were tested for two sizes of data.

1. A sentence with 5 words.
2. A paragraph text with 110 words.

**Results:**

1. Considering the data for a sentence the results of compositions are as follows:

<table>
<thead>
<tr>
<th>Service</th>
<th>Num of compositions</th>
<th>Number of devices</th>
<th>Time for composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanish to German</td>
<td>1</td>
<td>3</td>
<td>17.13 secs</td>
</tr>
<tr>
<td>Spanish to Italian</td>
<td>2</td>
<td>3</td>
<td>29.71 secs</td>
</tr>
<tr>
<td>Italian to English</td>
<td>3</td>
<td>3</td>
<td>40.82 secs</td>
</tr>
<tr>
<td>German to English</td>
<td>4</td>
<td>3</td>
<td>49.07 secs</td>
</tr>
<tr>
<td>Spanish to English</td>
<td>5</td>
<td>3</td>
<td>55.02 secs</td>
</tr>
</tbody>
</table>

Since the **time limit** for the completion of compositions was **2 minutes**, all of the compositions can be marked as **successful**.
2. Consider the paragraph text as the input data. This paragraph consists of 110 words. The result of compositions are as follows.

<table>
<thead>
<tr>
<th>Service</th>
<th>Num of compositions</th>
<th>Number of devices</th>
<th>Time for composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanish to German</td>
<td>1</td>
<td>3</td>
<td>25.96 secs</td>
</tr>
<tr>
<td>Spanish to Italian</td>
<td>2</td>
<td>3</td>
<td>57.95 secs</td>
</tr>
<tr>
<td>Italian to English</td>
<td>3</td>
<td>3</td>
<td>1 min 17 secs</td>
</tr>
<tr>
<td>German to English</td>
<td>4</td>
<td>3</td>
<td>1 min 53 secs</td>
</tr>
<tr>
<td>Spanish to English</td>
<td>5</td>
<td>3</td>
<td>2 min 13 secs</td>
</tr>
</tbody>
</table>

Here again the time limit was 2 minutes. Only 4 compositions could be completed within this time frame. Therefore the service composition with 5 levels can be marked as a failure for paragraph translation.

**Future Work:**

1. The framework developed currently functions on bluetooth. Support for other network protocols can be included in framework. For example, the network helper module can be extending by including the Bluetooth Low Energy protocol and Wi-Fi P2P protocols for peer to peer connectivity.
2. Mechanism to filter service compositions can be based on QoS parameters like cost, delay and execution time.
3. The message overhead in the network can be reduced if messages are forwarded to only those nodes that have a chance of coming in contact with the node that has the required service. Learnings from studies of mobility patterns of users would help improve the efficiency of the framework.
**Conclusion:**

From the results we can see that the time required for a composition depends on the number of intermediate compositions and the size of the input data. However, what is not apparent directly is that it also depends on the how frequently the nodes move about and come in contact with the devices that have the services. For example, in our test setup, if device 1 and device 3 were allowed to come in contact with each other, then number of hops required between services will be lower and eventually lead to quicker resolution of the requests for composed services.

A prototype application based on the above framework was developed for the Android Mobile OS and was successfully tested for service composition (in opportunistic networks) when two or more Android mobile devices have an opportunity to come in contact with each other. The design of the framework is completely generic and the corresponding modules seen the design can be implemented for any mobile system.

**References:**

