Designing Structured Peer-to-Peer Architectures for Mashups

Osama Al-Haj Hassan, Ashraf Odeh, Adi Maaita, and Aymen Abu-Errub

***Abstract*—Mashup platforms are popular because they provide end-users with a powerful tool to query the Web. The majority of these platforms are designed using centralized architectures or loosely coupled distributed architectures. The lack of structure in these platforms makes searching for mashups an expensive task. In this paper we present two structured peer-to-peer architectures for mashup platforms, namely, MashChord and MashCAN. These platforms increase the efficiency of searching for mashups. In addition, our design makes sure a mashup is hosted at several peers which strengthens the reliability of the system. Moreover, execution offloading feature is added to our design to support the functionality of each peer in the system.**

***Index Terms*—mashups, peer-to-peer, Web-2.0, structured, search, Chord, CAN.**

I. Introduction

TODAY'S Web focuses on topics such as semantic web [1], [2] and social networks [3]. In these types of applications, one of the most important features is personalization which is the hype of Web 2.0. One of Web 2.0 applications that offers personalization is mashup platforms. Mashup platforms empower end-users with a useful tool to search the Web in a personalized manner. They provide end-users with an interface to design data mashups. A mashup is basically a tree hierarchy that starts by fetching data from several data sources across the Web and proceeds by refining this data by using operators such as filter and truncate and ends by dispatching final result to end-user. One of the most popular mashup platforms is Yahoo Pipes [4]. An example of a mashup is shown in Figure 1 which fetches data from sport feeds, then filters the data based on containment of the term 'Rafa Nadal'.

Mashup platforms support personalization because they enable each end-user to create his own mashups. This is unlike web services which are designed for groups of people. Therefore, the number of mashups that a mashup platform hosts is expected to be very high. As a consequence, scalability issues arise for mashup platforms which requires careful attention.

Two existing architectures for mashup platforms are loosely distributed architecture and centralized architecture. Both architectures suffer from few drawbacks. In a loosely distributed architecture, peers of the network provide capability of designing and executing mashups. End-user uses this capability to design mashups of his own need. The advantage of using such an architecture is distributing load between network peers which increases the scalability of the system. However, the way communication takes place in loosely distributed architecture is by exchanging messages using flooding or some sort of random walks. This is an inefficient way of communication because if a user at one end of the network is looking for a mashup that exists at far end of the network, then messages have to be exchanged between neighbors gradually until the whole network is covered and the mashup is found. This is huge amount of messages needed for one search attempt. The problem exacerbates if the network is facing high traffic of search attempts.

Centralized architectures consist of one server which has the capability of designing and executing mashups. So, end-users use their machines to connect to this server, design and execute their mashups. The positive points of using this architecture is simplicity and a direct communication between client and server which results in a cheap search process. But, the system might fail if something wrong happens to the centralized server. In addition, the server might not be able to handle large number of users connecting to it. As a consequence, this architecture suffers from low reliability and scalability.

To overcome these problems, we extend our work in [5] by providing MashChord and MashCAN which are two designs for mashup platforms based on two popular peer-to-peer architectures, namely, Chord and CAN. Adding structure to a mashup platform increases its reliability and scalability. In addition, we add mashup execution offloading as a mean of relieving load on each peer in the network.

II. Literature Review

Our system represents mashup platforms over structured peer-to-peer topology. Therefore, our literature review will discuss the two aspects of mashup platforms and structured peer-to-peer networks.

A. Mashup Platforms

Mashup platforms are becoming very popular Web 2.0 applications. They have been investigated in literature. One famous mashup platform is Yahoo Pipes [4]. It is a platform that enables end-user to build mashups by providing a set of operators such as fetch, filter, and sort operators. The mashups built using Yahoo Pipes extract data from several types of data sources such as RSS and Atom feeds. MashMaker [6] is a mashup platform that enables end-users to extract data sources and populate them in a visual manner. Marmite [7] is a tool that helps end-users to aggregate several data sources and direct the end result to other files. This tool is implemented as a Firefox plug-in. MARIO [8] enables

Manuscript received January 2, 2014

O. Al-Haj Hassan is with the Department of Computer Science, Isra University, Amman, Jordan e-mail: osama.haj@ipu.edu.jo.

A. Odeh is with the Department of Computer Information Systems, Isra University, Amman, Jordan e-mail: ashraf.odeh@ipu.edu.jo.

A. Maaita is with the Department of Software Engineering, Isra University, Amman, Jordan e-mail: adimaaita@ipu.edu.jo.

A. Abu-Errub is with the Department of Computer Information Systems, Al-Ahliyya Amman University, Amman, Jordan e-mail: ayman_abuerrub@yahoo.com.

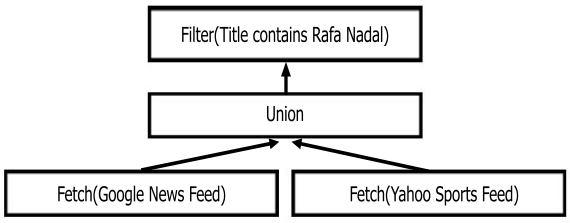

Fig. 1. Example of a mashup that which includes fetch, union, and filter operators

end-users to build mashups via choosing combination of tags from a cloud. In addition, MARIO executes mashups using an efficient execution plan. Karma [9] is a mashup platform that provides end-user with examples of mashups which he can alter to create his own mashups. Presto [10] provides a visual interface that facilitates the creation of secure enterprise mashups. DAMIA [11] discusses data aggregation for situational application. Jung [12] enables end-users to collaborate in order to build mashups. Di Lorenzo et al. [13] design a scheme that can be used to compare mashup platforms. The previous platforms rely on a centralized server architecture which makes them vulnerable against high workloads and that causes scalability issues. On the contrary, our system adopts a distributed model which avoids the single point of failure issue.

B. Structured Peer-to-Peer Networks

Peer-to-peer network is a well established area in literature. The main two types of them are Structured and Unstructured peer-to-peer networks. We refrain from using the unstructured topology because it uses flooding techniques as a search mechanism. Although the search process has been improved using for example random walks; this type of a topology still requires expensive search process. We use structured peer-to-peer topology in MashChord. Some of the most popular structured peer-to-peer platforms are Chord [14], CAN [15], and Pastry [16]. Chord [14] is a key lookup protocol that works by having a logical structured arrangement of peers and resources on a virtual ring topology. A hash function(SHA1) is used to generate identifiers for peers and resources. More information about Chord is provided in the next section because we adopt its topology and protocol in our work. CAN [15] is another structured peer-to-peer key lookup system. CAN arranges peers and resources in a virtual dimensional coordinate space such that each peer resides in a zone specific to it. Therefore, when a resource is mapped to a given zone, the peer responsible of that zone is the one that hosts and maintains that resource. Pastry [16] is a similar work to Chord where each peer is assigned a unique identifier from 128 bit space and Pastry protocol routes each message and key to the nodeID numerically closer to the given message key. Structured peer-to-peer networks can be used in different domains such as in [17] which surveys simulators built on top of structured and unstructured peer-to-peer networks. Another work [18] proposes a scheme that converts static network topology into a dynamic one built on top of structured peer-to-peer network. OE-P2RSP [19] is a structured peer-to-peer system built on top of Pastry. It adds enhancements over Pastry such as avoiding centralized object ID generation. It also uses objects group to make sure that objects that belong to the same group reside on the same node. The work in [20] targets the problem of free riding which happens when users make use of the peer-to-peer network without contributing with resources to the network.

Our work combines mashup platforms with structured peer-to-peer networks in order to come up with mashup platforms that benefit from efficient search process of structured peer-to-peer network and also benefit from distributed structure that avoids single point of failure in the system. These features fit well with the stringent scalability requirements of mashup platforms.

III. Mashup Representation

Each mashup is considered a tree of operator execution. This tree starts by fetch operators that fetch data from data sources distributed over the web. Then other operators such as filter, truncate, and sort operators process and refine the fetched data. After that, the final result is provided to the end-user.

Each mashup has a string representation in the system. This representation results from concatenating the representation of the operators that constitute the mashup. For example, the mashup in Figure 1 consists of 4 operators. The fetch operator to the left is represented as '11' where '11' is the ID of the data source 'Google News'. The fetch operator to the right is represented as '10' where '10' is the ID of 'Yahoo Sports' data source. The fetched data are combined using a union operator which has the representation '$SU|11|MU|10|EU$' where SU,MU,EU are separators between the two combined data sources '11' and '10'. The combined data is fed to a filter operator that filters data based on title containing keyword 'Rafa Nadal'. The filter operator is represented as '$15|07|30|Rafa\ Nadal$' where '15' is the ID of the filter operator, '07' is the ID of 'title' property, '30' is the ID of '$contains$' operation, and '$Rafa\ Nadal$' is the keyword on which filtering is executed. The representation of the operators in this example is shown in Figure 2. We also assign a representation for each subtree in mashups. This representation is the concatenation of representation of operators that are part of the subtree. In our example we have two subtrees. The first one consists of the two fetch operators in addition to the union operator. The second subtree is the whole mashup. The representation of the first subtree is '$SU|11|MU|10|EU$'. The representation of the second subtree is '$SU|11|MU|10|EU\#15|07|30|Rafa\ Nadal$' where the # symbol is used as a separator between the union and the filter operator representations. The representation of each subtree is shown in Figure 3.

IV. Structured P2P Mashup Architectures

In this section, we explain how to design MashChord and MashCAN which are mashup platforms that we build on top of two types of peer-to-peer architectures, namely, Chord and CAN (Content Addressable Network).

A. MashChord

In this subsection, we show the design of our mashup platform over Chord structured peer-to-peer network. First,

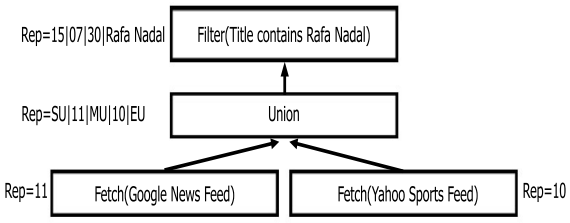

Fig. 2. Representation of each operator of the mashup in Figure 1

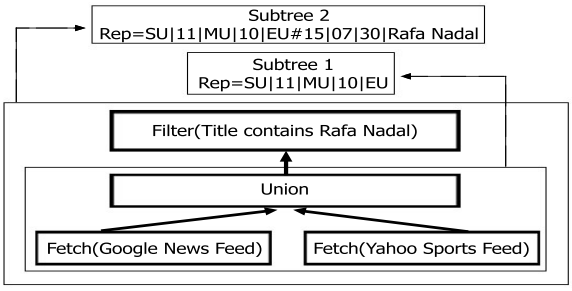

Fig. 3. Representation of each subtree of the mashup in Figure 1

we discuss the key features of Chord topology and protocol. Second, we provide the details of MashChord platform and how mashups are mapped to peers.

1) Chord: The first mashup platform we present is based on Chord [14]. Chord protocol arranges peers and resources on a virtual ring. The arrangement occurs by using SHA1 hash function. The hash function receives peer IP address as an input and it produces an identifier for that peer. Similarly, the hash function takes resource key as an input and it generates an identifier for that resource. The resulting identifier of a peer represents its location on the virtual ring. The concept 'successor' is important in Chord. The successor of an identifier 'k' is the peer with identifier 'k' or the peer that immediately follows 'k' on the ring (clockwise). For example, in Figure 4 successor(2)=2 because there is a peer with identifier 2. Also, successor(6)=0 because there is no peer with identifier 6 and the peer that immediately follows identifier 6 on the ring (clockwise) is 0. Given the concept of 'successor', the way resources are assigned to peers is simple. A resource 'k' is assigned to a peer successor(k). For example, resource with identifier 3 is hosted at peer 4 because successor(3)=4.

Searching for a given resource happens in the following way. Each peer has a finger table which contains several entries of the form {peer, peer_interval, successor_of_peer}. Each entry simply specify three things for a given peer. First, a peer identifier. Second, what interval this peer covers of the ring. Third, what is the successor of that peer. In Figure 4, suppose peer 2 is looking for resource 5. Using its finger table, peer 2 tries to find out successor(5). Unfortunately, the successor of identifier 5 is not found among finger table entries. Therefore, peer 2 finds the interval that contains 5 in the finger table. This interval is [4,6) and it is found in the second entry of the finger table. Based on that entry we see that successor(4)=4. Since 4 precedes 5, peer 2 contacts peer 4 asking for successor(5). Now, finger table of peer 4 indicates that successor(5)=0. Therefore, peer 4 informs peer 2 that peer 0 is the peer that is supposed to host resource 5 if it exists. As a result, peer 2 contacts peer 0 to find out whether it hosts resource 5. Accordingly, peer 0 returns the answer of the search query (Yes/No) to peer 2.

This is a brief description of Chord protocol that we adopt in our work. More details about Chord can be found in [14].

2) MashChord Mechanism: Our system consists of several peers. These peers are logically connected via a Chord ring as described in subsection IV-A1. In our platform, each peer has mashup components that enable the peer end-users to fully design, execute, and host mashups. The mashup execution component is responsible of executing mashups. The mashup user interface component is used by end-users to design new mashups and see the result of executing mashups. The offloading component coordinates with the mashup execution component to manage executing part of mashups on other peers. The search component is responsible of following the structure of the network to find mashups that satisfy end-users criterion.

The resources of our system are mashups. As explained in section III each operator/subtree of a mashup has a string representation. We start by explaining how a string representation is converted to a Chord identifier. The representations of operators/subtrees of mashup in Figure 1 are shown in Figures 2 and 3. First, a string representation is passed as an input to SHA1 hash function which results in a hexadecimal representation. Second, we supply the hexadecimal representation to a simple function which converts it to a decimal number. Third, the decimal number is divided by 2^m where m is the number of bits used to represent Chord identifiers. The remainder of the division process would be a Chord identifier. For example, the string representation of subtree1 is found in Figure 3. Supplying this representation to SHA1 function results in a hexadecimal number which is then converted to a decimal number. In our Chord example, we have 8-identifier Chord ring (0–7) which can be represented by at most 3 bits (000–111). Therefore, in our case $m = 3$. When the decimal number of subtree1 is divided by 2^3. The remainder of the division process is 6. Accordingly, subtree1 is assigned the identifier 6. Calculating the identifier of subtree1 is illustrated in Figure 5. Based on the previous discussion, the identifier is generated based on Equation 1 where 'R' is the string representation of the operator or subtree. The same process is repeated for each operator/subtree of the mashup which results in identifiers shown in the second column of Figure 6.

$$\textbf{identifier} = To\_Decimal(SHA1(R))\ mod\ 2^m \quad (1)$$

Now, we describe the mechanism to host resources (mashups) on peers of Chord ring. Regardless of Chord, the first peer responsible of hosting a mashup is the same peer at which the mashup is created. So, if an end-user at peer 0 created a mashup, that mashup would be hosted at the same peer. As an initial solution, that mashup is hosted on another peer in the network. This peer is decided as follows. As we previously mentioned, the mashup string representation is converted to a Chord identifier. Assume the resulting identifier is 'k'. So, successor(k) gives us the identifier of the peer responsible of hosting that mashup. This initial solution has a major drawback which is the inability of our system to satisfy partial matching queries. Usually, when an end-user executes a search query, he issues a search query that finds mashups

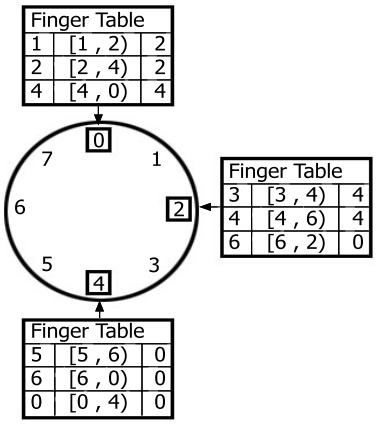

Fig. 4. A 8-identifier Chord ring with 3 peers

Action	Result
R=Subtree1 Rep	SU|11|MU|10|EU
R_1 = SHA1(R)	76202d7ed080c3d8fd8fc1f3e65e33986a0788f6
R_2=ToDecimal(R_1)	67437849800915921443216988947573292441992531378 2
R_3=R_2 mod 2^3	6

Fig. 5. Generating identifier 6 for subtrees1

containing certain subtree (Partial Matching). For example, the mashup illustrated in Figure 1 has string representation '$SU|11|MU|10|EU\#15|07|30|Rafa\ \ Nadal$'. When this representation is converted to a Chord key, the identifier 4 is the result. Accordingly, successor(4)=4 which indicates that peer 4 is going to host that mashup. Now suppose the end-user at peer 2 is looking to find mashups containing subtree1 indicated in Figure 3. Clearly, the mashup in conversation contains the desired subtree. So, this mashup is supposed to be returned as a result of the search query. Unfortunately, Chord protocol supports only exact matching queries. So, when the mashup representation of the subtree1 is converted to a Chord identifier, the resulting identifier is 6 and successor(6)=0 which indicates peer 0 (not peer 4 which actually hosts the mashup).

This leads us to think of a variation of this scheme which supports the partial matching operation. In the updated scheme, we state that a mashup is hosted on the following peers.

- The peer that is initially used to create the mashup.
- Each peer indicated by Chord protocol resulting from mapping all operators/subtrees of the mashup.

We explain this in the following example. First, assume the mashup in Figure 1 is created by peer 0 which in turn hosts that mashup. In addition, that mashup contains 6 operators/subtrees shown in Figures 2 and 3. The mashup representation for each operator/subtree is shown in the same figures. We convert the string representation for each operator/subtree to its corresponding Chord identifier. The result is identifiers 3,4,5,5,6 and 6 shown in the first column of Figure 6. Accordingly, successor(3)=4, successor(4)=4, successor(5)=0, and successor(6)=0 which indicates that the

Operator/Subtree	Identifier	Successor(identifier)
Fetch(Yahoo Sports)	5	0
Fetch(Google News)	3	4
Filter(Title contains Rafa Nadal)	5	0
Union	6	0
Subtree1: Union; Fetch(Google News) Fetch(Yahoo Sports)	6	0
Subtree2: Filter(Title contains Rafa Nadal); Union; Fetch(Google News) Fetch(Yahoo Sports)	4	4

Fig. 6. Operators and Subtrees mapped to Chord identifiers

mashup in conversation is also going to be hosted at peers 0 and 4. The successor for each identifier of operators/subtrees is found in the third column of Figure 6. Clearly, peer 0 is the peer on which the mashup is originally created, so, the mashup is not going to be duplicated on the same peer. Figure 7 shows that the mashup is hosted at peers 0 and 4.

Now, suppose the end-user at peer 2 issues a search query looking for mashups that contain the filter operator in Figure 1. The search process is performed as follows.

- The filter operator representation is converted to a Chord identifier which is 5.
- Peer 2 searches its finger table looking for successor(5). This information is not found in the finger table.
- Peer 2 finds that 5 falls in the interval [4,6) which is in the second entry of its finger table.
- The second entry of the finger table shows that successor(4)=4. Since 4 precedes 5 on the Chord ring, peer 4 is contacted looking for successor(5).
- Peer 4 finger table shows that successor(5)=0. So, Peer 4 contacts peer 0 asking whether it hosts a mashup with the desired filter operator.
- Peer 0 truly hosts such a mashup; and therefore a 'Yes' answer combined with the mashup is sent to peer 2.

Since MashChord is built on top of Chord protocol. Chord [14] states that a search process only requires $O(log\ N)$ messages where N is the number of peers in the network. This is much cheaper cost compared to flooding technique where number of messages increases exponentially as search progresses. This is why depending on Chord search mechanism makes searching for mashups in MashChord efficient.

B. MashCAN

In this subsection, we show how can a peer-to-peer mashup platform be implemented on top of CAN. We start by explaining the basics of CAN protocol. Then, we demonstrate how to apply it to our mashup platform.

1) CAN: The second mashup platform we introduce is based on CAN [15]. CAN arranges peers and identifiers on a virtual d-dimensional Cartesian coordinate space. For simplicity, we will assume 2 dimensional coordinate space

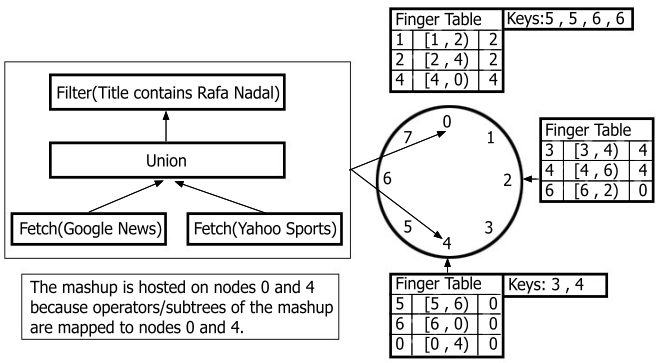

Fig. 7. Mapping process resulted in peers 0 and 4 hosting the mashup in Figure 1

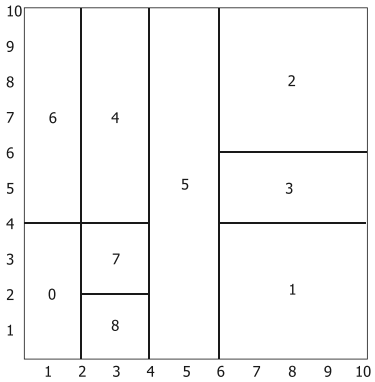

Fig. 8. A 2 dimensional CAN space with 9 zones controlled by 9 peers

such as the one in Figure 8 in which we have 9 peers occupying 9 zones. Each zone on the coordinate space can be represented by its center point (x,y). A zone can be occupied by only one peer. Notice that a neighbor peer for peer 'a' is a peer that shares at least part of 1 edge with peer 'a'. For example, in the previous figure, neighbors of peer 7 are peers 0, 4, 5,and 8. Regardless of the technique used in assigning a peer to a zone, if a new peer is to be placed in a zone while that zone is already occupied, then the zone splits into two zones such that the previous owner of the zone retains half of the zone and the new peer occupies the other half.

The way peers and identifiers are assigned to zones is as follows. One way to assign peers to zones is by simply placing them randomly on zones. Another way is by using a hash function for each dimension such that the values coming from the hash functions represent a point in the coordinate space which leads to a specific zone. For example, given a 2 dimensional space, we have a hash function for the x-coordinate and a hash function for the y-coordinate. Each hash function takes peer IP address as an input and it generates a value that maps to the designated coordinate. In other words, for some new peer assume the output of the first hash function is 3.0 and the output of the second hash function is 5.0. This means that the peer will take control of the zone in which the point (3.0,5.0) resides. This happens to be the same zone that peer 4 already occupies. As a consequence, that zone splits into two zones such that peer 4 retains half of the zone and the other peer takes over the other half.

Now, peers will host resources and the way CAN maps resources to peers is as follows. The same two hash functions explained earlier will receive resource key as an input and each hash function will generate the corresponding coordinate value for that resource. Accordingly, in a 2 dimensional space, assume the first hash function generates the x-coordinate value 8.0 and the second hash function generates the y-coordinate value 7.0. This means that the resource in conversation will be hosted at the peer that controls the zone in which the point (8.0,7.0) resides. According to the previous figure this peer is 2.

The search process for a given resource occurs in the following manner. First, it is important to know that each peer in CAN keeps a routing table in which each neighbor peer center point and coverage zone is saved. The search process depends on a simple greedy approach to come closer to where the resource is hosted. Suppose that peer 'n' initiated a search for resource 'r'. peer 'n' is represented by the center point of its zone (x,y) and the resource 'r' is represented by coordinate space point (x1,y1). peer 'n' computes the Euclidean distance between point (x1,y1) and the center point of each of the neighboring peers. Then peer 'n' simply forwards the search process to the peer in the zone that resulted in the minimum distance because it is supposed to be closer to the zone that hosts resource 'r'. This process is repeated until the peer that is supposed to host resource 'r' is reached.

2) MashCAN Mechanism: Our mashup platform consists of several peers logically arranged via CAN. Each peer has the capability of hosting, designing, and executing mashups.

In our implementation of MashCAN, we use a two dimensional coordinate space for ease of representation. We utilize SHA1 hash function for the x-coordinate and SHA256 function for the y-coordinate. We assume that each coordinate has the range [0,10). In other words, peer and resource identifiers takes a floating point number starting from zero and ending at 10 exclusively. Also, we place peers randomly to their zones. In our running example, we have 9 peers that control 9 zones which can be shown in Figure 8.

Now, we will explain how a resource key can be converted to a CAN coordinate point in MashCAN.

- SHA1 and SHA256 hash functions will receive as input the key for the subtree/mashup.
- The output of each hash function is a hexadecimal number.
- Each hexadecimal number is converted to its corresponding decimal equivalent.
- Each decimal number is divided by 10^{d-1} such that d is the number of digits in the decimal number.
- This way each decimal number is converted to a floating point number that falls in the interval [0,10).

As an example, Figures 9 shows how subtree1 key is converted to x-coordinate value and Figure 10 shows the same for y-coordinate value. In other words, converting a resource key to x-coordinate and y-coordinate values is computed based on equations 2 and 3 respectively. When the same process is applied to each subtree of the mashup in Figure 1, we end up with coordinate space points shown in Figure 11. The same figure also shows the zone in which each point resides.

Action	Result
R=Subtree1 Rep	SU|11|MU|10|EU
R_1 = SHA1(R)	76202d7ed080c3d8fd8fc1f3e65e33986a0788f6
R_2=ToDecimal(R_1)	67437849800915921443216988947573292441992531378 2
x-coord=$R_2/10^{d-1}$	6.7

Fig. 9. Generating x-coordinate for subtrees1. Note that d is the number of digits in the decimal number

Action	Result
R=Subtree1 Rep	SU|11|MU|10|EU
R_1 = SHA256(R)	b1c2419f2d8a66171bb21476c368611a54b05e334b20d3893182d5aa8dea209c
R_2=ToDecimal(R_1)	804025954347452058506610507637634716312336140172761063059099655903981659833 88
y-coord=$R_2\backslash 10^{d-1}$	8.0

Fig. 10. Generating y-coordinate for subtrees1. Note that d is the number of digits in the decimal number

$$\mathbf{x\_ccordinate} = \frac{To\_Decimal(SHA1(R))}{10^{d-1}} \quad (2)$$

$$\mathbf{y\_ccordinate} = \frac{To\_Decimal(SHA256(R))}{10^{d-1}} \quad (3)$$

In MashCAN, a mashup will be hosted by the following peers.

- The peer on which the mashup is initially designed.
- Peers that each subtree of the mashup maps to according to CAN protocol.

We will explain this by showing an example of mapping a subtree/mashup to a peer in MashCAN. Assume peer 6 designs the mashup in Figure 1. This means that peer 6 will host that mashup. In addition, the mashup will be hosted by peers controlling zones in which each subtree coordinate point resides. As shown in Figure 11 subtree1 has coordinate point (6.7,8.0) which resides in zone controlled by peer 2. Therefore, peer 2 will also host the mashup. By looking at Figure 11 we see that different subtrees has coordinate points (1.0,3.3), (1.3,3.6), (1.1,6.5), (6.7,8.0), (6.7,8.0), and (1.1,1.6). These points resides in zones controlled by peers 0, 0, 6, 2, 2, and 0 respectively. As a result, peers 0, 2, and 6 are the peers that host the mashup in conversation (Figure 12). Clearly, peer 6 is not going to duplicate the mashup because it was originally designed at that peer. In order for mashup hosting to take place, peer 6 will issue a host request via its neighbors to peers 0 and 2.

Now, we will explain how a given peer initiates a search process for a subtree/mashup in MashCAN. Assume peer 1 issues a search for mashups that contain the filter operator shown in Figure 1. The search process advances as follows.

- The filter operator representation is converted to a CAN coordinate space point as explained earlier. As Figure 11 shows, this point is (1.1,6.5).
- Peer 1 asks their neighbors (peers 3 and 5) if point (1.1,6.5) lies within their zone. The answer is No.
- Peer 1 finds which neighbor is closer to the zone in which point (1.1,6.5) resides by computing Euclidean distance between the point (1.1,6.5) and the center point for each neighbor zone, namely (8.0,5.0) and (5.0,5.0). The resulting distance is 7.0 and 4.1 respectively.
- Since distance between peer 5 and point (1.1,6.5) is the minimum, then peer 1 forwards the search request to peer 5 because it is supposed to be one step closer towards the zone in which point (1.1,6.5) resides.
- peer 5 repeats the same process and forwards search request to peer 4.
- peer 4 finds that the point (1.1,6.5) resides within the zone of its neighboring peer 6.
- peer 6 truly hosts a mashup that contains the filter operator as a subtree, therefore the answer Yes is returned to peer 1 which originally initiated the search process.

MashCAN is built on top of CAN. CAN [15] states that a search process requires only $O(d*n^{\frac{1}{d}})$ steps where 'd' is the number of dimensions and 'n' is the number of peers. This is drastically more efficient than the exponential complexity of flooding search approach.

V. Scalability and Reliability

Scalability and reliability are two important features for a networking system. MashChord and MashCAN are scalable and reliable because of three reasons. First, they follow a structured peer-to-peer architecture that does not rely on a given peer as the core of the system. This structured type of peer-to-peer network has minimal search overhead due to depending on structure to map resources to peers. MashChord and MashCAN rely on Chord and CAN respectively which are well established works that handle peer joins and leaves efficiently. We will not discuss peer joins and leaves as they are described in Chord protocol [14] and CAN [15] and they are not the main focus of this paper.

Second, the scalability and reliability of our system is extended by the nature of our mashup mapping protocol. Remember that a mashup is hosted by several peers in the network which are found by generating identifiers in the case of MashChord and coordinate points in the case of MashCAN for each subtree of a given mashup. If one of those peers decides to leave the system, the system functionality is not affected because the mashup is also hosted on several other peers. For example, suppose that an end-user is looking for mashups that contain subtree1 shown in Figure 3. If such mashups exist, they would be hosted on 2 peers, namely, 0 and 4. Consequently, if one of those peers fails or voluntarily leaves the network, the mashups can still be found on the other peer.

Third, we further enhance the scalability and reliability of our system by designing 'Execution Offloading' mechanism. One possible scenario is that one peer is busy performing operations of its own. The end-user at that peer wants to execute certain set of mashups. As explained in section IV-A2, the peer has the necessary mashup components to execute the mashups. But, its CPU is busy performing other work. We can exploit the fact that a mashup is hosted at several peers

Operator/Subtree	(x,y) coordinate	Zone
Fetch(Yahoo Sports)	(1.0,3.3)	0
Fetch(Google News)	(1.3,3.6)	0
Filter(Title contains Rafa Nadal)	(1.1,6.5)	6
Union	(6.7,8.0)	2
Subtree1: Union; Fetch(Google News) Fetch(Yahoo Sports)	(6.7,8.0)	2
Subtree2: Filter(Title contains Rafa Nadal); Union; Fetch(Google News) Fetch(Yahoo Sports)	(1.1,1.6)	0

Fig. 11. Operators and Subtrees mapped to CAN coordinate points

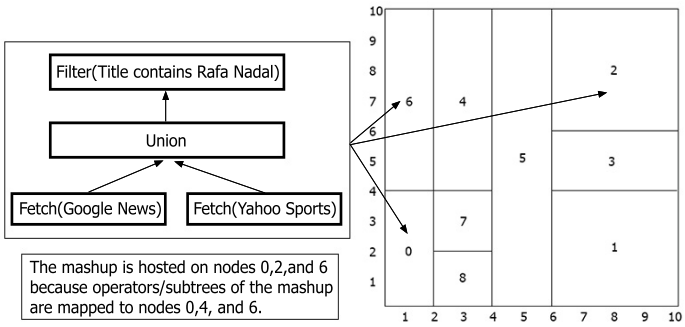

Fig. 12. Mapping process resulted in peers 0, 2, and 6 hosting the mashup in Figure 1

to offload all/part of mashup execution to other peers that host that same mashup. One thing to mention here is that each peer declares a percentage indicating how busy it is. We add this piece of information to Chord finger tables and MashCAN routing tables. Now, when a given peer wants to offload part of its mashup execution to another peer, it chooses the peer with minimum busy percentage. Also, each peer has an offloading percentage which represents the percentage of mashups to offload their execution to other peers. The previous features aid towards a scalable and reliable mashup platform.

VI. Efficient Mashup Access

Efficiency is an important feature for any computer system. If the application is a network oriented one, then efficiency becomes extra important due to the many variants affecting its functionality. There has been attention towards efficiency in literature [21], [22]. We need to take care of efficiency of peers in MashChord and MashCAN because each peer might end up hosting lots of mashups. That might lead to inefficient mashup access. For example, suppose a peer received request to execute a mashup that it currently hosts; then it is the responsibility of that peer to search the list of mashups and subtrees it hosts for the desired mashup. If the number of hosted mashups is large, then sequential searching process would take long time to conclude.

Therefore, we add indexing structure to each peer that aids towards fast mashup access. The type of index we use is a B-tree index where index keys are the representation of mashups and subtrees which was previously explained in section III. The keys are added to the index based on their lexicographical order. Accessing B-tree index requires $O(log\ n)$ steps which is very low in contrast with the $O(n)$ steps required for sequential search. As a result, using this mashup index increases the efficiency of searching for mashups.

VII. System Evaluation

We use simulation to evaluate MashChord and MashCAN. Our topology consists of 128 peers, 12800 total mashups originally created at peers. Number of operators per mashup is varied between 4 and 8. Offloading percentage for peers is varied between 10% and 90%. Busy percentage for peers is varied between 10% and 90%. The peers we use in our simulation are extracted from 2012 Internet topology measured by DIMES [23] and [24].

In the first experiment, we show the effect of mashup execution offloading. We pick one peer randomly and we vary offloading percentage for this peer between 10% and 90%. Then we measure the execution time spent by that peer. Figure 13 shows that the execution time spent by that peer decreases as the offloading percentage increases. This makes sense because the peer has to execute a subset of mashups as the execution of the rest of mashups is offloaded to other peers. A related experiment is shown in Figure 14 where declared busy percentage is changing between 10% and 90%. We notice that as busy percentage increases the execution time decreases. This is because a high busy percentage makes peers reject execution offloading requests coming from other peers.

In the next experiment, we measure the average number of mapped mashups per peer when the number of operators per mashup increases. Here, we are not pointing to the original mashups created at each peer. We only focus on the number of mashups that are hosted on other peers due to operator/subtree mapping to Chord identifiers. Here, we vary number of operators to be between 4 and 8. Figure 15 shows that average number of mapped mashups per peer increases as number of operators per mashup increases. When number of operator per mashup increases, the number of subtrees per mashup increases. Therefore, we have more subtrees that are mapped to Chord identifiers. As a consequence, these additional identifiers cause mashups to be hosted on more peers.

In the last experiment, we test the effectiveness of our b-tree index. Remember that a peer hosts a number of mashups. When that peer receives a search request, the fist thing it performs is to look for the requested mashups/subtrees within the mashups it hosts. This process can be done sequentially or using the B-tree index that we explained in section VI. Figure 16 shows that number of steps needed to conclude a search process within MashChord and MashCAN

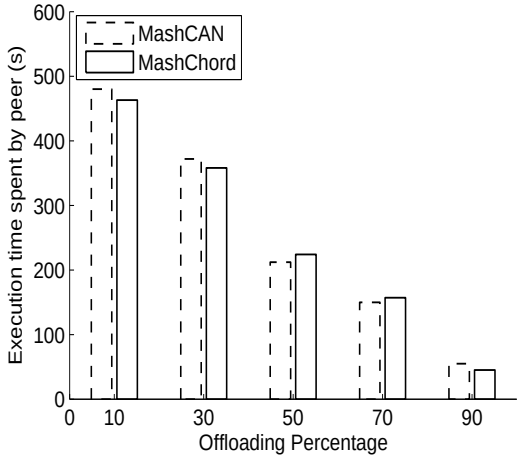

Fig. 13. Execution time spent by a peer when offloading percentage varies

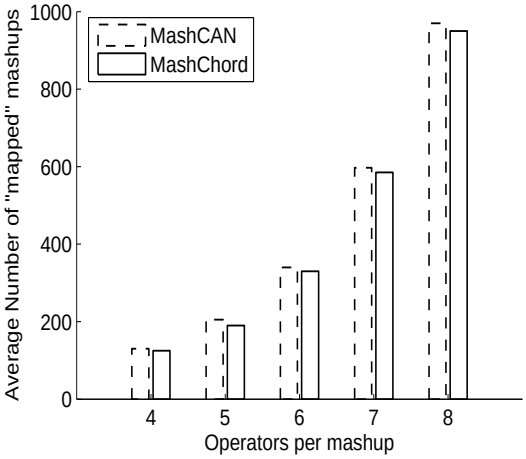

Fig. 15. Average number of hosted mashups per peer when number of operators per mashup varies

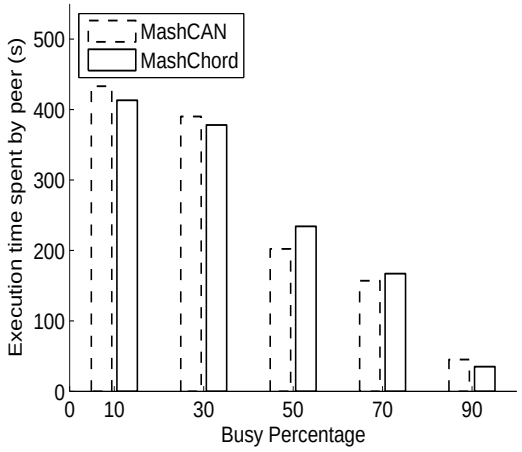

Fig. 14. Execution time spent by a peer when busy percentage varies

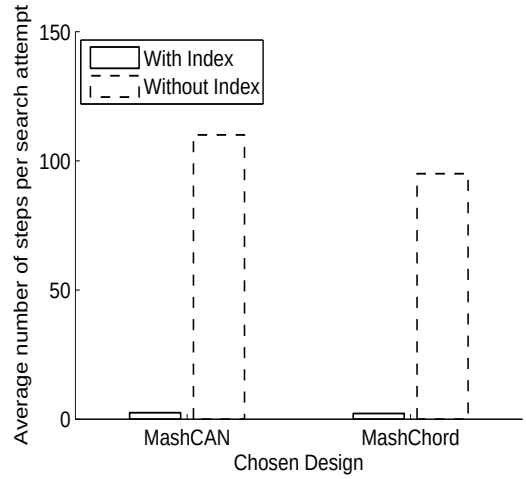

Fig. 16. Average number of steps per search attempt with and without using the b-tree index

is tremendously low when the b-tree index is used. This is a result of $O(log\ n)$ performance of the b-tree index versus $O(n)$ performance of sequential search.

The previous experiments shed light on the importance of mashup execution offloading in our system. They also pointed out that the increase in number of operators per mashup would increase the load of hosting mashups on peers. In addition, the effectiveness of using B-tree index for search efficiency is plotted in the experiments.

VIII. Conclusion

One of the popular Web 2.0 applications is mashup platforms. Current mashup platforms rely on loosely-distributed architectures and centralized architectures. This raises scalability and reliability issues for mashup platforms. In this paper, we built MashChord and MashCAN which are mashup platforms designed on top of structured peer-to-peer architectures. MashChord and MashCAN have low search overhead. In addition, we add execution offloading feature to our system which enhance the functionality of each peer. Moreover, we use an indexing structure that increases the efficiency of local search for mashups at each peer. These features helps towards better efficiency, reliability, and scalability in MashChord and MashCAN.

References

[1] F. KIMURA, H. URAE, T. TEZUKA, and A. MAEDA, "Multilingual translation support for web pages using structural and semantic analysis," *IAENG International Journal of Computer Science*, vol. 39, no. 3, pp. 276–285, Aug. 2012.

[2] O. Kazik and R. Neruda, "Ontological modeling of meta learning multi-agent systems in owl-dl," *IAENG International Journal of Computer Science*, vol. 39, no. 4, pp. 357–362, Nov. 2012.

[3] R. Neruda and O. Kazik, "Formal social norms and their enforcement in computational mas by automated reasoning," *IAENG International Journal of Computer Science*, vol. 39, no. 1, pp. 80–87, Jun. 2012.

[4] Yahoo Inc., "Yahoo pipes," http://pipes.yahoo.com/, 2007.

[5] O. Al-Haj Hassan and A. Odeh, "Mashchord: A structured peer-to-peer architecture for mashups based on chord," in *Proceedings of The World Congress on Engineering and Computer Science 2013*, vol. 1. Hong Kong: International Association of Engineers, 2013, pp. 1–6.

[6] R. J. Ennals and M. N. Garofalakis, "Mashmaker: mashups for the masses," in *ACM SIGMOD international conference on Management of data*, 2007, pp. 1116–1118.

[7] J. Wong and J. Hong, "Making mashups with marmite: towards end-user programming for the web," in *SIGCHI conference on Human factors in computing systems*, 2007, pp. 1435–1444.

[8] A. Riabov, E. Bouillet, M. Feblowitz, Z. Liu, and A. Ranganathan, "Wishful search: interactive composition of data mashups," in *WWW*. New York, NY, USA: ACM, 2008, pp. 775–784.

[9] R. Tuchinda, P. Szekely, and C. Knoblock, "Building mashups by example," in *International Conference on Intelligent User Interfaces*, 2008, pp. 139–148.

[10] JackBe Corp., "Presto enterprise mashups," http://www.jackbe.com/products/presto, 2011.

[11] IBM Corp, "Damia," http://services.alphaworks.ibm.com/damia/, 2007.
[12] J. J. Jung, "Collaborative browsing system based on semantic mashup with open apis," *Expert Systems with Applications*, vol. 39, no. 8, pp. 6897–6902, Jun. 2012.
[13] G. Di Lorenzo, H. Hacid, H.-y. Paik, and B. Benatallah, "Data integration in mashups," *SIGMOD Rec.*, vol. 38, no. 1, pp. 59–66, Jun. 2009.
[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/383059.383071
[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 161–172. [Online]. Available: http://doi.acm.org/10.1145/383059.383072
[16] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, ser. Middleware '01. London, UK, UK: Springer-Verlag, 2001, pp. 329–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=646591.697650
[17] S. Naicken, A. Basu, B. Livingston, S. Rodhetbhai, and I. Wakeman, "Towards yet another peer-to-peer simulator," in *Proceedings of the International Working Conference in Performance Modelling and Evaluation of Heterogeneous Networks*, 2006.
[18] T. Jacobs and G. Pandurangan, "Stochastic analysis of a churn-tolerant structured peer-to-peer scheme," *Peer-to-Peer Networking and Applications*, vol. 6, no. 1, pp. 1–14, 2013. [Online]. Available: http://dx.doi.org/10.1007/s12083-012-0124-z
[19] S. ZHANG and H. Jun, "Building structured peer-to-peer resource sharing platform using object encapsulation approach," *TELKOMNIKA Indonesian Journal of Electrical Engineering*, vol. 11, no. 2, pp. 935–940, 2013.
[20] M. Karakaya, I. Korpeoglu, and O. Ulusoy, "Free riding in peer-to-peer networks," *Internet Computing, IEEE*, vol. 13, no. 2, pp. 92–98, 2009.
[21] N. P. Khanyile, J.-R. Tapamo, and E. Dube, "An analytic model for predicting the performance of distributed applications on multicore clusters," *IAENG International Journal of Computer Science*, vol. 39, no. 3, pp. 312–320, Aug. 2012.
[22] W. NEW, K. WEE, Y. WEE, and C.-O. WONG, "Wimax: Performance analysis and enhancement of real-time bandwidth request," *IAENG International Journal of Computer Science*, vol. 40, no. 1, pp. 20–28, Feb. 2013.
[23] Y. Shavitt and E. Shi, "Dimes: let the internet measures itself," *ACM SIGCOMM*, vol. 35, no. 5, pp. 71 – 74, May 2005.
[24] Shavitt, Yuva, "Dimes," http://www.netdimes.org/, 2009.

Osama Al-Haj Hassan obtained his BS in Computer Science from Princess Sumayya University for Technology (PSUT). He also received his MS in Computer Science from New York Institute of Technology (NYIT). He obtained his PhD in Computer Science from University of Georgia (UGA). Currently, he is an Assistant Professor at Isra University in Jordan. His research interests are in distributed systems, web services, Web 2.0, caching and replication techniques, peer-to-peer networks and event based systems.

Ashraf A. Odeh is an Assistant Professor in Computer Information System at Isra University-Jordan. He received a BSc degree in Computer Science in 1995 and MSc degree in Information Technology in 2003. With a Thesis titled "Visual Database Administration Techniques", He received PhD from department of Computer Information System in 2009 with a Thesis titled "Robust Watermarking of Relational Database Systems". He is interested in image processing, Watermarking, Relational Database, E-copyright protection, E-learning and E-content. He has submitted a number of conference papers and journals. Also he has participated in a number of conferences and IT days.

Adi Maaita obtained his BS in Computer Science from The University of Jordan. He also received his MS in Computer Science from New York Institute of Technology (NYIT). He obtained his PhD in Computer Science from The University of Leicester. Currently, he is an Assistant Professor at Isra University in Jordan. His research interests are in web engineering, genetic algorithms, machine learning, and object-oriented software engineering.

Aymen Abu-Errub received his Ph.D. degree in Computer Information Systems from the Arab Academy for Banking and Financial Sciences, Jordan, in 2009. He is an assistant professor in Faculty of Information Technology in Al-Ahliyya Amman University, Jordan in Computer Information Systems department and then in Networks and Information Security department. He has published journal papers in security, information retrieval, and in risk management fields. He also participated and published his researches in scientific conferences.