

MashChord: A Structured Peer-to-Peer Architecture for Mashups Based on Chord

Osama Al-Haj Hassan, Ashraf Odeh, and Anas Abu Taleb

Abstract—Mashups are key category of Web 2.0 applications. Due to their personalization property, mashup platforms suffer from scalability and efficiency issues. Most mashup platforms are based on either centralized or loosely distributed architectures and that causes several deficiencies when searching for mashups. This paper presents MashChord—a structured peer-to-peer (P2P) architecture for mashups based on Chord. Naturally, Chord supports exact matching search queries. We show how we customize this to provide partial matching search queries in MashChord. MashChord architecture increases the efficiency of searching for mashups. We utilize this structured architecture to incorporate several features into MashChord that further improve the functionality of our platform such as offloading mashup execution to peers.

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

I. INTRODUCTION

MASHUPS are an icon of Web 2.0 applications. A mashup is defined as a Web service designed by end-user. The purpose of a mashup is to fetch data sources distributed over the Web, process the fetched data and further refine it based on several operators such as filter, truncate, and sort operators. An example is a mashup that fetches data from Yahoo sports feed and Google news feed, then combines these two data sources, and filters the combined data based on title containing 'Rafa Nadal'. This mashup is shown in Figure 1.

The main difference between a mashup and a Web service is that mashups are designed by end-users while Web services are designed by developers to fulfill the need of a specific group of people. In other words, mashups aid towards more personalized user experience because each end-user can design his own mashups based on his own needs.

The personalization property of mashups comes on the expense of scalability problems. Since each end-user can design his own mashups, it is expected that the number of mashups hosted by mashup platforms is very high. Therefore, mashup platforms have to be designed carefully so that scalability issues are alleviated.

One of the existing mashup architectures is the centralized architecture. In this architecture, a mashup server is responsible of hosting and executing mashups. So, end-users use the mashup application deployed on the server to design, execute, search, and save mashups. This architecture is simple. Searching for mashups is a straight forward cheap process because all mashups are hosted at one machine

(server). However, it is a single point of failure in the system which jeopardizes the system functionality. In addition, this architecture is not the appropriate architecture to use given the scalability issues explained before because one server cannot handle the increasing number of mashups and their requests.

Another architecture is a loosely distributed one which consists of many nodes. Nodes of this architecture represent clients and mashup processing nodes. The advantage of this architecture is the distributed load on nodes. However, being a loose architecture causes a main drawback which is the vast amount of messages exchanged between nodes in order for the system to work correctly. For example, suppose one end-user started a search process to find mashups that satisfy a certain criterion. This search process will expand through the network via flooding technique or its variants which causes huge number of messages to be generated between nodes. All that is performed for one search process, the problem exacerbates when executing high number of search attempts.

In order to fix the previously mentioned problems, we present the design of a mashup platform on top of a structured network architecture that helps in distributing load between nodes. The structured part aids towards a faster search process.

II. LITERATURE REVIEW

Our system represents a mashup platform 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 [1]. 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. Mash-Maker [2] is a mashup platform that enables end-users to extract data sources and populate them in a visual manner. Marmite [3] 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 [4] enables end-users to build mashups via choosing combination of tags from a cloud. In addition, MARIO executes mashups using an efficient execution plan. Karma [5] is a mashup platform that provides end-user with examples of mashups which he can alter to create his own mashups. Presto [6] provides a visual interface that facilitates the creation of

Manuscript received June 28, 2013; revised July 19, 2013.

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. Abu Taleb is with the Department of Computer Science, Isra University, Amman, Jordan e-mail: anas.taleb@ipu.edu.jo.

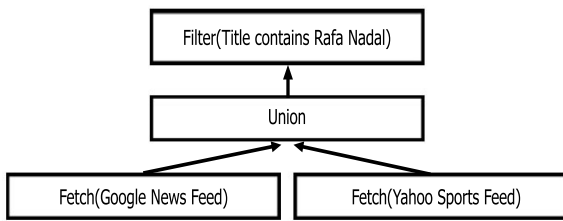


Fig. 1. Example of a mashup that which includes fetch, union, and filter operators

secure enterprise mashups. DAMIA [7] discusses data aggregation for situational application. Jung [8] enables end-users to collaborate in order to build mashups. Di Lorenzo et al. [9] 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 [10], CAN [11], and Pastry [12]. Chord [10] 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 [11] 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 [12] is a similar work to Chord where each node 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 [13] which surveys simulators built on top of structured and unstructured peer-to-peer networks. Another work [14] proposes a scheme that converts static network topology into a dynamic one built on top of structured peer-to-peer network. OE-P2RSP [15] 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 [16] 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.

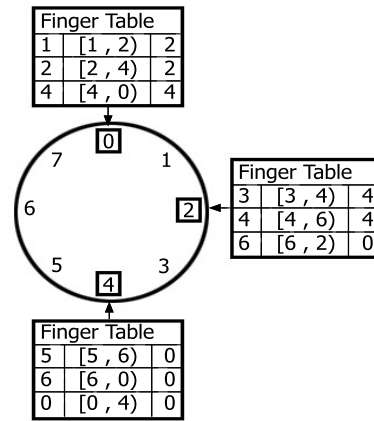


Fig. 2. A 8-identifier Chord ring with 3 nodes

Our work (MashChord) combines mashup platforms with structured peer-to-peer networks in order to come up with a mashup platform that benefits from efficient search process of structured peer-to-peer network and also benefits 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. MASHCHORD

In this section, we show the design of our mashup platform over Chord structured P2P network. First, we discuss the key features of Chord topology and protocol. Second, we explain mashup representation in our system. Third, we provide the details of MashChord platform and how mashups are mapped to peers.

A. Chord

Our system is based on Chord [10]. 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 2 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 2, suppose peer 2 is looking for resource 5. Using its finger table, peer 2 tries to find out successor(5). Unfortunately,

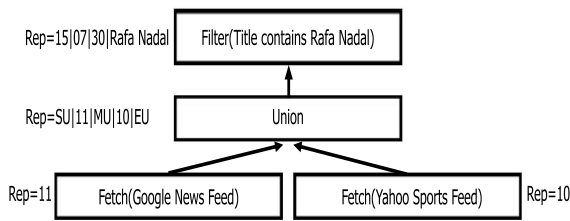


Fig. 3. Representation of each operator of the mashup in Figure 1

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 [10].

B. 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 3. 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

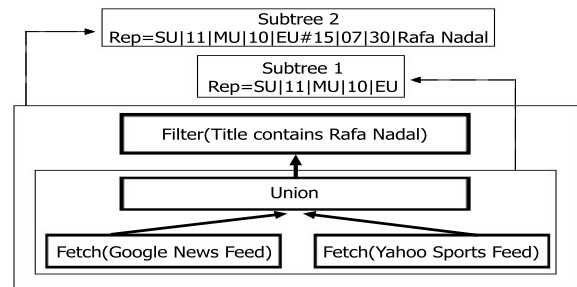


Fig. 4. Representation of each subtree of the mashup in Figure 1

the filter operator representations. The representation of each subtree is shown in Figure 4.

C. MashChord Mechanism

Our system consists of several peers. These peers are logically connected via a Chord ring as described in subsection III-A. 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-B 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 3 and 4. 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 4. 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.

$$\text{identifier} = To_Decimal(SHA1(R)) \bmod 2^m \quad (1)$$

Now, we describe the mechanism to host resources (mashups) on peers of Chord ring. Regardless of Chord, the

Action	Result
$R = \text{Subtree1 Rep}$	SU 11 MU 10 EU
$R_1 = \text{SHA1}(R)$	76202d7ed080c3d8fd8fc1f3e65e33986a0788f6
$R_2 = \text{ToDecimal}(R_1)$	674378498009159214432169889475732924419925313782
$R_3 = R_2 \bmod 2^3$	6

Fig. 5. Generating identifier 6 for subtrees1

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 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 4. 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 3 and 4. 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 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,

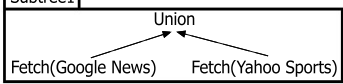
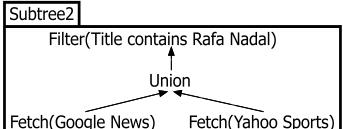
Operator/Subtree	Identifier	Successor(identifier)
Fetch(Yahoo Sports)	5	0
Fetch(Google News)	3	4
Filter(Title contains Rafa Nadal)	5	0
Union	6	0
Subtree1 	6	0
Subtree2 	4	4

Fig. 6. Operators and Subtrees mapped to Chord identifiers

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 [10] 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.

IV. SCALABILITY AND RELIABILITY

Scalability and reliability are two important features for a networking system. MashChord is scalable and reliable because of three reasons. First, it follows a structured P2P architecture that does not rely on a given node as the core of the system. This structured type of P2P network has minimal search overhead due to depending on structure to map resources to peers. MashChord relies on Chord which is a well established work that handles node joins and leaves efficiently. We will not discuss peer joins and leaves as they are described in Chord protocol [10] and they are not the main focus of this paper.

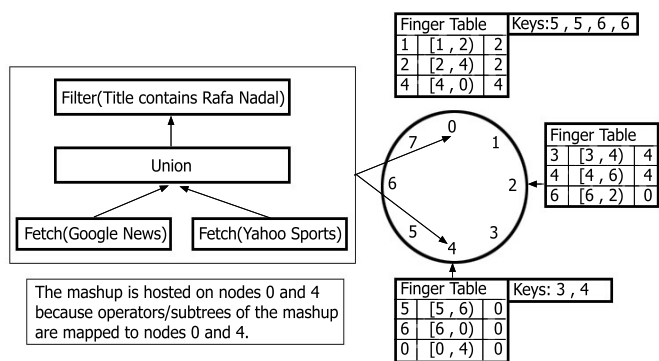


Fig. 7. Mapping process resulted in peers 0 and 4 hosting the mashup in Figure 1

Second, the scalability and reliability of our system is extended by the nature of our mashup mapping protocol. Remember that a mashup is hosted on several peers in the network which are found by generating identifiers 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 4. 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 III-C, 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 in several peers 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 the Chord finger table. 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 represent the percentage of mashups to offload their execution to other peers. The previous features aids towards a scalable and reliable mashup platform.

V. SYSTEM EVALUATION

We use simulation to evaluate MashChord system. Our topology consists of 128 nodes, 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%. The peers we use in our simulation are extracted from 2012 Internet topology measured by DIMES [17] and [18].

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 8 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.

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 9 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.

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.

VI. FUTURE WORK

This work provides a design of a mashup platform over Chord structured peer-to-peer network. We have two things in mind that we intend to investigate in future. First, it is noticed from the running example we explained in this paper that several mashup operators/subtrees might result in the same identifier and hosted on the same peer. Such a peer that receives a search query does not have a problem resolving this issue. This happens by searching all mashups that it hosts until the required mashup is found. The point here is that this sequential search to all mashups it hosts can be costly. Therefore, we intend to investigate using indexing structures based on the keys of each peer. This is expected to minimize local search cost for peers.

Second, we mentioned earlier that each peer declares a percentage indicating how busy it is. If this percentage is low, this means that the peer is capable of receiving many offloaded mashup execution requests from other peers. Consider a scenario where a peer deliberately declares a high busy percentage with the intention to not receiving mashup execution requests from other peers. The peer plays a negative role in this case which is considered a sort of the popular free riding problem in which certain peers use the network without contributing much to it. Consequently, we intend to investigate how to prevent peers from declaring false busy percentage. This can be achieved by adding a certain part of the peer software that checks how busy the CPU is and therefore checks if the declared busy percentage makes sense. We can also achieve this by using a formula that allows a peer to offload mashup execution only if it maintains a minimum average busy percentage per week/month.

VII. CONCLUSION

Mashup platforms are flourishing as an important Web 2.0 applications. Existing mashup platforms suffer from scalability problems. In this work, we proposed MashChord—A scalable and reliable mashup platform over Chord. MashChord uses structured P2P topology to facilitate executing

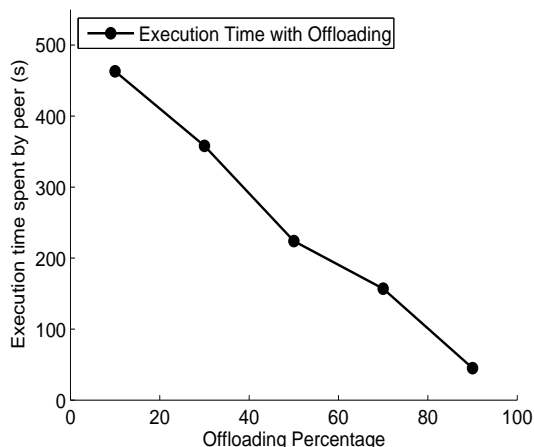


Fig. 8. Execution time spent by a peer when offloading percentage varies

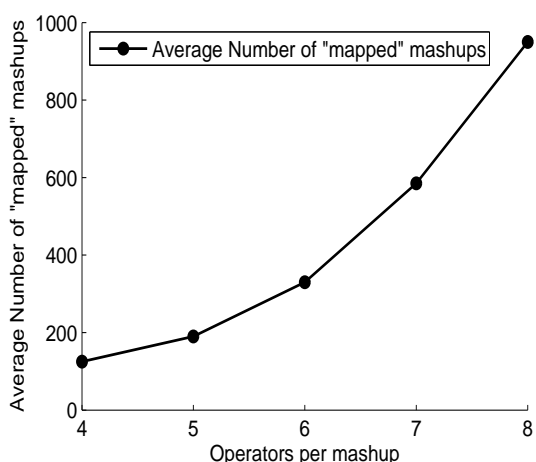


Fig. 9. Average number of hosted mashups per peer when number of operators per mashup varies

mashups and searching for them. Following the structured P2P mechanism improves search efficiency. Also, hosting a mashup at several peers allows end-user to perform partial matching search over Chord which originally supports exact matching. We explained how execution offloading improves the efficiency of MashChord.

REFERENCES

[1] Yahoo Inc., "Yahoo pipes," <http://pipes.yahoo.com/>, 2007.
[2] 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.
[3] 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.
[4] 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.
[5] R. Tuchinda, P. Szekely, and C. Knoblock, "Building mashups by example," in *International Conference on Intelligent User Interfaces*, 2008, pp. 139–148.
[6] JackBe Corp., "Presto enterprise mashups," <http://www.jackbe.com/products/presto>, 2011.
[7] IBM Corp., "Damia," <http://services.alphaworks.ibm.com/damia/>, 2007.
[8] 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.

[9] 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.
[10] 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>
[11] 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>
[12] 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>
[13] 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.
[14] 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>
[15] 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.
[16] 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.
[17] Y. Shavitt and E. Shi, "Dimes: let the internet measure itself," *ACM SIGCOMM*, vol. 35, no. 5, pp. 71 – 74, May 2005.
[18] Shavitt, Yuva, "Dimes," <http://www.netdimes.org/>, 2009.