

# CoMaP: A Cooperative Overlay-based Mashup Platform

Osama Al-Haj Hassan, Lakshmish Ramaswamy and John A. Miller

Computer Science Department, University of Georgia,  
Athens, GA 30602, USA  
{hasan,laks,jam}@cs.uga.edu

**Abstract.** Recently, mashups have emerged as an important class of Web 2.0 collaborative applications. Mashups can be conceived as personalized Web services which aggregate and manipulate data from multiple, geographically-distributed Web sources. Mashups, while enhancing personalization, bring up new scalability and performance challenges. The fact that most existing mashup platforms are centralized further exacerbates the scalability challenges. Towards addressing these challenges, in this paper, we present the design, implementation, and evaluation of *CoMaP* – a cooperative information system for mashup execution. The design of *CoMaP* is characterized by a scalable architecture with multiple cooperative nodes distributed across the Internet and possibly multiple controllers which plan and coordinate mashup execution. In our architecture, an individual mashup can be executed at several collaborative nodes with each node executing part of the mashup. *CoMaP* includes a unique mashup deployment scheme that decides which nodes would be involved in executing an individual mashup and what operators they would host. Also, *CoMaP* continuously adapts to overlay dynamics and to user actions such as creation of new mashups or deletion of existing ones. Furthermore, *CoMaP* possesses failure resiliency feature which is necessary for cooperative information systems. Our experimental study indicates that the proposed techniques yield improved system performance.

**Key words:** cooperative information systems, Web 2.0, mashup, collaboration

## 1 Introduction

Web 2.0 continues to evolve and grow at a tremendous pace. Web 2.0 applications are characterized by high degrees of personalization and social interaction, and they enable seamless information sharing and collaboration among end-users [15]. Mashups are one such class of popular Web 2.0 applications. Mashups can be conceptualized as personalized Web services that are created by end-users. Functionally, they fetch data from one or more Web sources, which would then be aggregated, manipulated and filtered as per user specifications, and the final results would be dispatched to the end-users. Yahoo pipes [17] and Intel

MashMaker [8] are among the popular mashup platforms. Mashup platforms are also considered a type of collaborative information systems that enable collaboration and sharing of data among end-users by allowing them to browse each other mashups and use them to build their own customized ones.

Mashups, while enabling high-degree of personalization, flexibility, and collaboration are also faced with unique scalability and performance limitations. First, unlike traditional Web services, mashups are designed by end-users. This implies that number of mashups hosted by a mashup platform is orders of magnitude higher than number of Web services hosted by a Web service portal. Second, mashups rely upon data from many different sources distributed across the Internet. These data sources vary widely in terms of their data characteristics, and reliability. Third, since mashups are developed by non-tech-savvy end-users, they may not be optimized from efficiency stand-point. These limitations are exacerbated by the fact that most existing mashup platforms are centralized.

This paper explores distribution as a mechanism to achieve scalability and performance of mashups. We present a dynamic cooperative mashup execution framework called *CoMaP*. *CoMaP* is based upon an overlay of nodes that collaborate to execute mashup process (either partial or complete). The collaboration between nodes is facilitated by a controller which also plans the execution of individual mashups. In designing *CoMaP*, we make three novel contributions.

- First, we present an efficient cooperative mashup architecture in which multiple nodes cooperate to execute mashups. Our architecture is distributed and mashup operators are spread across several nodes. The collaboration between mashup execution nodes is facilitated by a controller which also plans the execution of individual mashups.
- Second, we introduce a dynamic mashup distribution technique that is sensitive to the locations of the various data sources of an individual mashup as well as the destinations of its results. Our technique progressively optimizes the network load in the overlay and the latency of mashup execution.
- Third, we handle failure resiliency issues in our architecture through replicating nodes and replicating parts of mashup workflows.

We evaluate the design of *CoMaP* through series of experiments that show the effectiveness of our architecture and distribution technique.

## 2 Motivation and Challenges

In this section, we introduce background about mashups and current functionality of existing mashup platforms. In addition, we discuss motivations that led us to introducing *CoMaP* and challenges that we need to handle in our architecture. Mashups are popular because of their personalization property that enables each end-user to design his own mashups based on his own needs as opposed to a Web service which is dedicated to a group of end-users. The operators involved in mashups can be operators for fetching data from data sources distributed across the Web. They can also be data processing operators such as filter operators.

Notice that each mashup operator  $Op_z$  might have multiple parents and multiple children where children operators are the source of input to  $Op_z$  and parent operators are the sinks of output of  $Op_z$ .

In addition to Yahoo pipes [17], other mashup platforms exist in literature such as MARIO [11], DAMIA [7], Marmite [16], and MashMaker [8]. These platforms work in several ways such as providing a cloud of tags from which end-users build their mashups, or allowing end-users to work on data sources using a visual interface. To the best of our knowledge, these platforms are based on a centralized server through which end-users create and execute mashups.

Since each end-user has the privilege of designing his own mashups, mashup platforms typically host large numbers of mashups and experience high mashup request rates. Because of that, a centralized mashup platform faces an increasing pressure and might not be able to keep up with increasing amount of end-users requests which raises a scalability problem. Also, a centralized mashup platform does not consider the geographical location of end-users and data sources; this implies that some end-users might observe high delays. In addition, having a centralized mashup platform implies that the centralized server is a single point of failure. The previous three points motivates the need for a distributed mashup platform where mashup execution takes part on several cooperative nodes.

## 2.1 Challenges

Distributing mashup execution requires the collaboration of distributed nodes in an overlay that faces network dynamics; therefore, we need to handle several challenges in distributing mashup execution.

Since we are designing a distributed system, what type of cooperation is needed between network nodes? This is important to guarantee a complete and correct mashup execution. Also, should a mashup be executed on one node or multiple nodes? Executing a mashup on multiple nodes forms a way of parallel execution. Further, what are the parameters based on which a node is selected to execute the whole mashup or part of it? Parameters such as communication links delay, bandwidth and nodes loading, should be considered.

The next challenges are related to handling dynamics of the system; what happens in the case of changing network parameters? For example, communication links delay and bandwidth between nodes is changing due to factors such as congestion. Also, node loading is changing as nodes get more mashups to execute. Therefore, a mashup that is being executed on some nodes might have to be reassigned to different nodes to adapt to changing network parameters. Also, what if more than one end-user share the same mashup or part of it? A mashup execution plan is initially designed based on a number of end-users requesting it. When more end-users request the same mashup, the mashup execution plan might change based on the geographical location of the new end-users. Further, what happens if network nodes fail? Certain recovery method should be applied to make sure system functionality is not affected?

Towards addressing the previous challenges, we proceed by introducing our distributed mashup platform *CoMaP*.

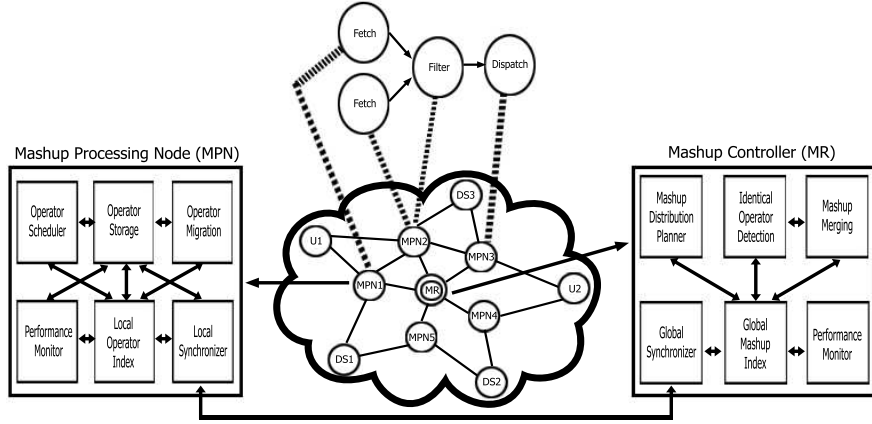
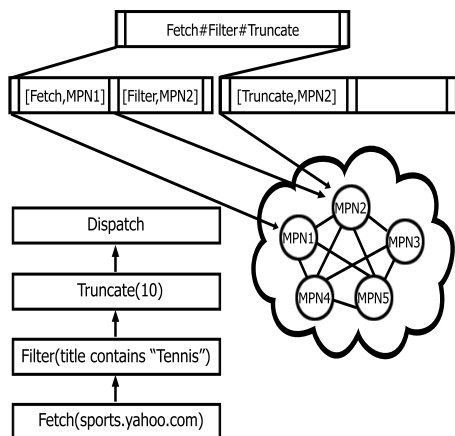


Fig. 1. CoMaP Architecture

### 3 CoMaP Architecture

Figure 1 illustrates *CoMaP*'s high-level design architecture. *CoMaP* is based upon an overlay of mashup processing nodes *MPNs*, we refer to this set of nodes as  $MPNSet = \{MPN_1, MPN_2, \dots, MPN_L\}$ . These nodes are distributed across the Internet, and they collaborate to execute mashup workflows. A mashup controller *MR* plans the execution of each mashup. It also coordinates the activities of various processing nodes involved in the execution of a mashup. A set of end-users  $USet = \{U_1, U_2, \dots, U_N\}$  interact with *CoMaP*. Each of those end-users can design and submit his own mashups to *MR* using a mashup designer. Note that each *MPN* and end-user in *CoMaP* can tell what its coordinates are by probing a set of nodes geographically distributed over the Web (Landmarks). Landmarks [9] cooperate among each other to deliver coordinates for the node that probed them. We refer to the set of mashups that *MR* receives as  $MpSet = \{Mp_1, Mp_2, \dots, Mp_M\}$ . Operators that form those mashups are referred to as  $OpSet = \{Op_1, Op_2, \dots, Op_Z\}$ . The previous entities are connected with a set of communication links where each communication link  $LNK_{e,f}$  connects node  $e$  with node  $f$ . Each communication link  $LNK_{e,f}$  has a delay  $DeLNK_{e,f}$  and a bandwidth  $BndLNK_{e,f}$ .

Each *MPN* is responsible for executing a set of workflows as determined by *MR*. An individual workflow might correspond to an entire mashup or part of it. A mashup workflow is essentially a tree of operators. When executing a workflow, one *MPN* may fetch data from external sources or it may receive partially processed data from other processing nodes which would have executed earlier parts of the mashup. The results are dispatched either to the end-user (if no further processing is needed for the mashup) or to another *MPN* (if mashup execution is not yet complete). Executing a mashup on several nodes yields better efficiency and scalability. For example, if a mashup consists of two fetch operators



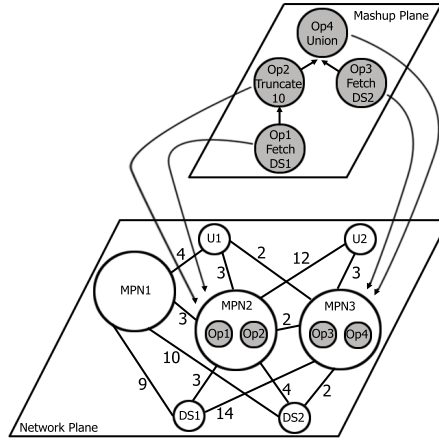
**Fig. 2.** A mashup stored in B+ tree which points to *MPNs* where operators are deployed

that fetch data from two different data sources, then assigning each operator to a different node facilitates parallel execution.

As indicated in Figure 1, each *MPN* comprises of several components. Mashup workflows assigned to the processing node are stored in the operator storage, and are indexed by the operator index. The local scheduler makes the workflow scheduling decisions. The performance of each processing node is locally monitored, summary of which is communicated to the global performance monitor (located at *MR*) periodically.

End-users interact with the system through *MR* which they get to know upon joining the system. The set of interactions include creation and deployment of new mashups and deletion of existing ones. When an end-user sends a new mashup to *MR* to be executed, *MR* first checks whether the newly created mashup shares operator sequences with existing mashups. If so, *MR* modifies the mashup to utilize the results from the existing operator sequences so that duplicate computations are avoided. The global mashup index aids the detection of shared mashup operators. This index is similar to the index that we introduced in *AMMORE* [2] except that the index introduced in *AMMORE* is based on a centralized architecture. Therefore, the global index used by each *MR* is extended from *AMMORE* index so that it contains information about where each operator is deployed in the network. Figure 2 shows an example of the global operator index.

Once *MR* is done with shared operators detection, it uses its mashup distribution planner to decide where (on which execution nodes) an incoming mashup would be executed, and if multiple nodes are involved in executing a mashup what part each would execute. The mashup distribution planning considers several factors including the mashup structure, the locations of the data sources and end-users,



**Fig. 3.** The operator placement problem in CoMap

and the current performance of the overlay. The global performance monitor at *MR* interacts with the local performance monitors at individual *MPNs*, and it maintains a global snapshot of the overlay performance. The global synchronizers coordinate the activities of the *MPNs* involved in executing a mashup. Figure 3 illustrates a mashup being executed on two *MPNs*. Next, we explain our technique for distributing mashup execution and how it can adapt to network changes.

## 4 Planning Distributed Mashup Execution

This section formally describes the distributed mashup execution problem. After that, we explain how to plan the deployment of mashup operators in the overlay network and how they are executed in a distributed fashion.

### 4.1 Problem Statement

Figure 3 demonstrates the operator placement problem. We have a plane of mashups and a plane of network nodes, data sources, and end-users. System cost differs based on where operators are placed in the overlay network. Therefore, our goal is to place each operator in a node in the network such that system cost is minimum. Towards solving this problem, we model the distributed mashup execution problem as an optimization problem. In *CoMaP*, we consider delay, bandwidth, and nodes load as metrics for computing the cost of executing operators in different places in the overlay network.

When a certain operator is executed on a given node, the cost of that execution is partitioned into computation cost and communication cost. Computation cost

results from processing time of executing the operator on the node and communication time results from transmitting operators execution output to the next set of nodes that host the operators coming next in the mashup workflow. Suppose we have an operator  $Op_z$  that belongs to mashup  $Mp_i$ , created by end-user  $U_j$ , and hosted by  $MPN_k$ . The input of this operator is coming from the operators which are executed before  $Op_z$ , we refer to this set of operators as  $PrvOpSet$  and we refer to some operator in this set as  $Op_q$ . We refer to the output size of  $Op_q$  as  $OS_q$ . The computation cost of  $Op_z$  deployed on  $MPN_k$  can be formulated as the summation of the size of the output data of each operator in  $PrvOpSet$  in kilobytes divided by time needed by  $MPN_k$  to process each kilobyte of the data  $PT_k$ . Thus,  $CompTime_{z,k} = \sum_{q=1}^Q \frac{OS_q}{PT_k}$ .

Once  $Op_z$  operator finishes execution, it needs to send its output to the  $MPNs$  that host the next set of operators that are expecting input from  $Op_z$ . We refer to this set of  $MPNs$  as  $NxtMPNSet$  and we refer to some  $MPN$  in this set as  $MPN_r$ . Communication time resulted by operator  $Op_z$  deployed on  $MPN_k$  can be formulated as the size of operator  $Op_z$  output in kilobytes  $OS_z$  divided by outgoing communication link bandwidth  $BndLNK_{k,r}$  between  $MPN_k$  and each  $MPN_r$  in  $NxtMPNSet$ , the result is added to outgoing link's communication delay per unit of data  $DeLNK_{k,r}$  between  $MPN_k$  and each  $MPN_r$ . Therefore,  $CommTime_{z,k} = \sum_{r=1}^R (\frac{OS_z}{BndLNK_{k,r}} + DeLNK_{k,r})$ .

As a result, cost of execution of operator is the combination of computation and communication costs. Notice, that operators are executed multiple times according to their request rate, so, the total cost for an operator has to be multiplied with the operator's request rate  $RT_z$ . Therefore, cost of executing operator  $Op_z$  on  $MPN_k$  becomes  $C_{z,k} = RT_z \times ((\sum_{q=1}^Q \frac{OS_q}{PT_k}) + \sum_{r=1}^R (\frac{OS_z}{BndLNK_{k,r}} + DeLNK_{k,r}))$ .

System cost results from the combination of delay resulting from 1) End-users  $USet$  submitting their mashups to Mashup Controller  $MR$  2)  $MR$  performing operators merging on requested mashups 3)  $MR$  assigning mashup operators to  $MPNSet$  4) Executing mashup operators  $OpSet$  on  $MPNSet$ . Cost in 4 can be optimized, therefore, our target in this optimization problem is to place operators  $OpSet$  on  $MPNSet$  such that total cost of executing operators is minimum. Let  $Alloc_{z,k}$  be a  $\{0,1\}$  variable denoting that operator  $Op_z$  is deployed on  $MPN_k$  ( $Alloc_{z,k} = 1$ ), otherwise,  $Alloc_{z,k} = 0$ . Therefore, the optimization problem is to assign values to each  $Alloc_{z,k}$  variable such that  $\sum_{z=1}^Z \sum_{k=1}^L Alloc_{z,k} \times C_{z,k}$  is minimized while ensuring that the following constraints are not violated: 1) For an operator  $Op_z$ ,  $\sum_{k=1}^L Alloc_{z,k} = 1$ , this indicates that  $Op_z$  is deployed only on one  $MPN$ ; 2)  $LD_k \leq LdLimit_k$  which indicates that the load of  $MPN_k$  should not exceed its maximum allowed load.

A few naive approaches can be used to deploy mashup operators in overlay network, the first one is 'Random' deployment where mashup operators are distributed randomly on  $MPNs$ . Another approach is 'Destination' deployment where mashup operators are deployed on  $MPNs$  that are closest to the end-user who requested the mashup. The opposite approach is 'Source' deployment where mashup operators are deployed on  $MPNs$  that are closest to data sources that

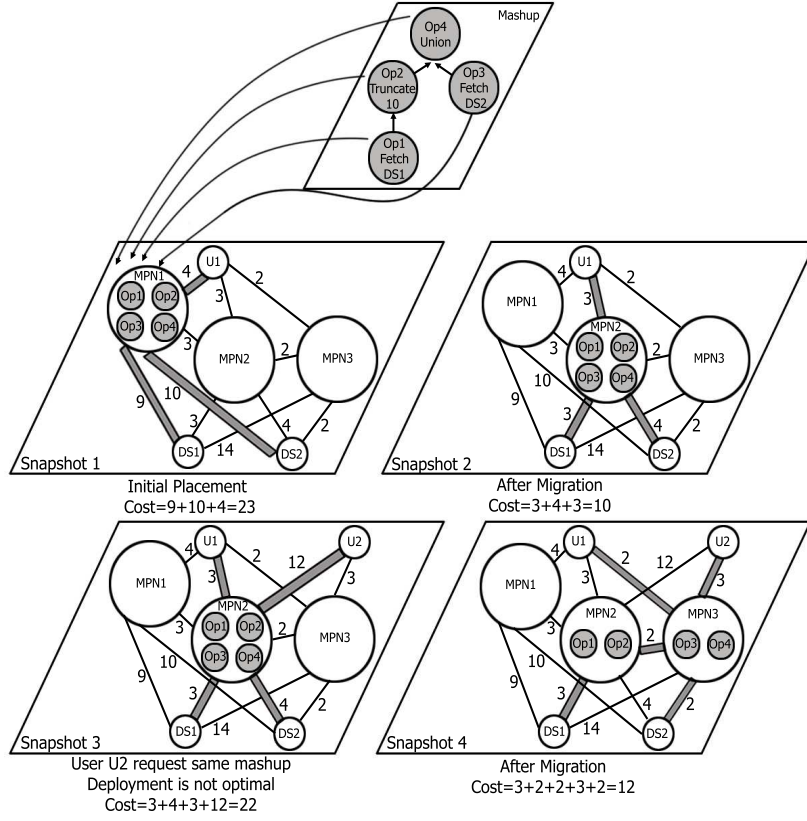


Fig. 4. A scenario of CoMaP operator placement

contribute to these operators. One more approach is the ‘*Optimal*’ deployment in which an exhaustive search is performed to deploy operators on *MPNs* that yield the minimum cost. The ‘*Optimal*’ approach is expected to produce the best result but its running time is exponential due to its exhaustive search nature.

## 4.2 Our Scheme - DIMA

This subsection describes our approach for deploying mashup operators. Our approach is a two-stage optimization process where initial operator deployment is performed in the first stage and a migration process takes place in the second stage. We introduce a running example represented in Figure 4 throughout our discussion. In this example, a mashup consisting of four operators is to be deployed on *MPNs*. The mashup first uses a fetch operator ( $Op_1$ ) to pull data from data source  $DS_1$ , then it uses  $Op_2$  to truncate 10 items from the result of fetching data. Second, the mashup uses a fetch operator ( $Op_3$ ) to fetch data from data source  $DS_2$ . Finally, the results of  $Op_2$  and  $Op_3$  are combined us-

ing a union operator ( $Op_4$ ), and the final result is dispatched to end-user ( $U_1$ ). Figure 4 consists of 4 parts representing in-order snapshots of the system. Link delays on the figure represent the final delay (in seconds) resulting from taking propagation delay, bandwidth, and data source sizes into consideration. For the sake of clarity in this figure, we assume all  $MPNs$  have the same processing power. One thing to mention is that geographical location of a node is reflected by its location in the layout. For example, by looking at part 1 of the figure, we can see that  $MPN_1$  is closer to  $U_1$  than  $MPN_2$ . Communication links that contribute to system cost are shaded in grey.

**Stage1: Initial Deployment** Mashup Controller  $MR$  has information about all  $MPNs$  in the system, including load of  $MPNs$  and their coordinates in the network distance graph. Such information is transferred to  $MR$  by  $MPNs$  upon joining the network. The load information for  $MPNs$  is then updated as  $MR$  deploys mashup operators on  $MPNs$ . Once an end-user sends his mashup to  $MR$ , the coordinates for the end-user machine is also sent to  $MR$ .

In this stage, when the mashup arrives to  $MR$ ,  $MR$  uses end-user coordinates and  $MPNs$  coordinates to compute Euclidian distance between each pair of end-user and  $MPN$ . This distance is then used to estimate delay of sending data from  $MPN$  to end-user. Each operator is then initially deployed on the  $MPN$  that has the minimum delay from the end-user. In our running example, part 1 of Figure 4 shows the initial deployment stage for the four operators. Notice that delay in this stage is only computed based on coordinates (Euclidean distance). Here, since  $MPN_1$  is geographically closer to  $U_1$  than  $MPN_2$  and  $MPN_3$ , all operators are initially deployed on  $MPN_1$ .

This initial operator deployment may not always be optimal for the following reasons. First, this stage depends on Euclidian distances to estimate delays which is not accurate as relying on current network conditions. In part 1 of our running example, actual link delays do not comply with geographical location of nodes. This can happen because some nodes reside on fast links, others might reside on slower links. It can also happen because of congested links. Second, operator deployment in this stage is based on distances from end-users, if distances from end-users and data sources are taken into consideration; better deployment decisions might be achieved. In part 1 of our running example, delays between  $MPN_1$  and data sources  $DS_1$  and  $DS_2$  are not considered. However, distances from data sources are unknown at this stage because the coordinates of data sources are not known. The number of data sources on the Web is huge, this is why the system cannot store and maintain coordinates of that large number of data sources. Third, as we will explain in the next subsection, mashup operator deployment becomes sub-optimal when more end-users share mashups.

Once the initial placement of each operator is decided, the entry for that operator in the global mashup index is updated with the new deployment information that specifies at which  $MPN$  the operator is deployed. Although this initial step may not lead to optimal deployment decision, it is a good initial step. Next, the

optimization process seeks to improve deployment by going through a migration process.

**Stage2: Operator Migration** To minimize system cost, operators should migrate from the *MPN* on which they are deployed to a new *MPN* that leads to a better deployment decision. This migration process has to be fully distributed to preserve the scalability feature of *CoMaP*.

Each *MPN* has to decide if migrating operators to one of its neighbors decreases the total system cost; the total system cost is not available for *MPN* nodes due to the distributed nature of *CoMaP*. Basically, the total system cost is the summation of operators execution costs all over the network. So, if each *MPN* minimizes operator execution cost within its neighborhood, that will eventually decrease the total system cost.

To compute the amount of cost change resulting from migrating operators from *MPN<sub>i</sub>* to *MPN<sub>j</sub>*, first, we introduce the cost of a migration *state*. Each migration step has two states; *CurrentState* and *NewState* where the *CurrentState* represents hosting operators on *MPN<sub>i</sub>* (Before migration) and the *NewState* represents hosting operators on *MPN<sub>i</sub>* in addition to hosting migrated operators on *MPN<sub>j</sub>* (After migration). Therefore, we have two costs; *CurrentStateCost* and *NewStateCost* where the cost of each state includes the cost of hosting operators in that state; such that the cost of hosting an operator on some *MPN* is given as follows; 1) the cost of sending input to *MPN* by the nodes that host the children of this operator. 2) The cost of processing the input on *MPN*. 3) The cost of sending output by *MPN* to nodes that host parents of this operator. 4) The cost of communication between *MPN* and the end-users sharing the operator which is evaluated by pinging end-users machines. When *MPN<sub>i</sub>* is considering migrating operators to *MPN<sub>j</sub>*, it computes  $NetB = CurrentStateCost - NewStateCost$  and it also considers all direct neighbors *MPN<sub>j</sub>* (number of hops=1). After that, operators migrate to the neighbor with maximum positive *NetB* value. A positive *NetB* value indicates that this migration step leads to minimization of system cost.

Now, we demonstrate this stage in part 2 of our running example, *MPN<sub>1</sub>* is considering migrating all 4 operators to *MPN<sub>2</sub>*. In this case, *MPN<sub>1</sub>* uses ping pong messages to estimate cost of fetching data from *DS<sub>1</sub>* and *DS<sub>2</sub>*. Then, it uses ping pong messages to estimate cost of sending the result to *U<sub>1</sub>*. Accordingly,  $CurrentStateCost = 23$ . Now, *MPN<sub>1</sub>* asks its neighbor *MPN<sub>2</sub>* about the cost of hosting the 4 operators on it. Here, *MPN<sub>2</sub>* uses ping pong messages to estimate the cost of fetching data from *DS<sub>1</sub>* and *DS<sub>2</sub>* plus the cost of sending result to *U<sub>1</sub>*; then *MPN<sub>2</sub>* sends the result back to *MPN<sub>1</sub>*. Based on *MPN<sub>2</sub>* feedback, *MPN<sub>1</sub>* finds out that  $NewStateCost = 10$ . After that, *MPN<sub>1</sub>* calculates the net benefit ( $NetB = 13$ ) and decides to migrate all 4 operators to *MPN<sub>2</sub>*.

Notice that if no such **direct** neighbor with  $NetB > 0$  is found, *MPN<sub>i</sub>* widens its search process by looking at indirect neighbors where the number of hops = 2. This increase in the tested neighborhood helps ‘*DIMA*’ scheme to avoid local optima. At the same time, the maximum number of hops we use for the local

search process is 3, it is kept low so that *CoMaP* efficiency is not degraded. Also, this parameter can be set by the system administrator.

Part 3 of our running example considers the case when a new user  $U_2$  requests the same mashup which  $U_1$  initially requested. Because  $U_1$  and  $U_2$  share mashups, the previous operators deployment which is based on  $U_1$  requesting the mashup is not optimal any more. This happens because communication between  $MPN_2$  and  $U_2$  results in high delay. Therefore, migration is needed again. In this example,  $MPN_2$  is considering migrating  $Op_3$  and  $Op_4$  to  $MPN_3$  as a target neighbor.  $MPN_2$  repeats the same migration steps which results in  $NewStateCost = 12$ ,  $CurrentStateCost = 22$ , and  $NetB = 10$ . As a result,  $MPN_2$  decides that  $Op_3$  and  $Op_4$  should migrate to  $MPN_3$  which is reflected in part 4 of Figure 4.

When an operator migrates from  $MPN_i$  to  $MPN_j$ ,  $MPN_i$  informs the  $MPNs$  on which children and parent operators are deployed with such a change. In addition,  $MPN_i$  informs the mashup controller about the new change. The mashup controller in turn updates its mashup index with the new deployment decisions. Therefore, when new requests for mashups arrive to the mashup controller, the controller is able to consult the up to date mashup index to find out to which  $MPNs$  the mashup execution should be directed.

The migration process is performed periodically to ensure that *CoMaP* adapts to newly requested mashups and to changes in the number of end-users sharing operators. Moreover, the migration process needs to be executed periodically to adapt to changes in network links delay and bandwidth. Note that communication costs between  $MPNs$  is calculated such that delay and bandwidth values are the actual values of the links in the overlay network.

Notice that the cost of probing nodes done in the migration process is considered tolerable for the system because probing only occurs periodically when the migration process is performed. In addition, the migration process happens within each  $MPN$  locality which means that the migration process is performed smoothly without the system functionality being affected.

In the next section, we discuss failure resiliency which is an important quality for cooperative distributed information systems.

## 5 Failure Resiliency

The sources of failure in *CoMaP* could come from 1) Failure of Mashup Controller (*MR*) or 2) Failure of Mashup Processing Nodes (*MPNs*). Failure of a mashup controller is handled by replicating it which helps to avoid a single point of failure in the system. Among those controllers, one of them is the main controller and the other replicas are secondary controllers. All mashup controllers are identical to one another in terms of system information they possess and each one of them plays the same role in terms of receiving and handling end-user requests. However, the main controller plays slightly different role than secondary controllers in failure resiliency process. All mashup controllers are chosen to be distributed geographically in the network such that each controller serves the end-users closer to it.

To guarantee correct functionality of *CoMaP*, certain interaction between controllers is needed. For example, detecting shared operators requires that each controller knows about all operators in all mashups sent by end-users. Therefore, when one controller receives a mashup from an end-user, it directly sends the mashup information to all other controllers. This way, detecting shared operators becomes identical in all controller nodes. However, when a mashup is sent to a controller, that controller is the only one responsible of directing the execution of that mashup.

Another type of communication is needed between controllers in the case of controller failure. Basically, each controller has one identical list of nodes which specifies three pieces of information. First, who is currently the main controller? Second, what is the current set of secondary controllers? Third, what is the set of candidate nodes that can be replacements of secondary controllers? The main controller exchange heart beat messages with secondary controllers to ensure they are still alive. If the main controller did not receive a reply from one of the secondary controllers, it assumes it failed and responds by performing the following operations. First, it uses the candidate set to assign a new secondary controller. Second, its current state is duplicated on to the new secondary controller so that it can start operating. Third, it notifies all other secondary controllers about the failure of the old controller and the existence of the new replacement. Failure of the main controller is handled as follows, in case secondary controllers do not receive heart beat messages from the main controller, they assume it failed. Here, the current set of secondary controllers is used to recover from such a failure. Basically, what happens is that the current set of secondary controllers is ordered such that the first controller in the list has the responsibility of replacing the main controller when it fails. In this case, the previously mentioned secondary controller (new main controller) performs the following operations. First, it eliminates itself from the current secondary controllers list. Second, it propagates the change in this list to all other controllers. Third, it announces itself as the new main controller to all other secondary controllers. The introduction of coordinator replicas only causes one change on ‘*DIMA*’ operator deployment scheme. When an operator migrates from  $MPN_i$  to another  $MPN$ ,  $MPN_i$  contacts the coordinator in its area to inform it about operator migration. That coordinator in turn distributes the information to all other coordinators. So far, we discussed failure within controllers, now, we discuss failure of Mashup Processing Nodes (*MPNs*). *MPNs* exchange heart beat messages with direct neighbors, if one *MPN* did not receive a reply from one of the neighbors, it assumes failure of that neighbor and reports the failure to the controller in its area. Once the controller receives the failure notification, it reallocates the operators which used to be hosted by the failed *MPN* to a new *MPN*. This new allocation is propagated to the main controller and all secondary controllers. Moreover, if failure occurred to one of the mashup processing nodes (*MPNs*), then the effect of this failure is alleviated by replicating operators. If one *MPN* hosting an operator fails, then the operator can still be accessed through its replica. Since a huge number of operators exist in *CoMaP*, we cannot replicate

each one of them. So, we select a percentage of the most overloaded *MPNs* in the system and we replicate their operators. This percentage is selected by the system administrator based on observed system performance. When one *MPN* replicates an operator to another *MPN*, it also informs the controller in its area of the replication process, and that controller propagates this change to all other controllers.

## 6 Experimental Evaluation

We use simulation to perform our experiments. The goal of experiments is to evaluate ‘*DIMA*’ approach by showing its effect on *CoMaP* performance. Also, we discuss the effect of applying failure resiliency on our system.

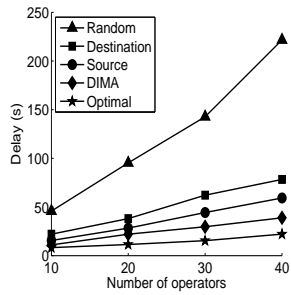
### 6.1 Experimental Setup

*CoMaP* environment simulates several operators which is similar to yahoo pipes, these operators are Fetch, Filter, Sort, Union, Truncate, Tail, Sub-Element, Reverse, Unique, and Count. Our system consists of 4 mashup controllers, 100 data sources, 100 end-users, and a number of *MPNs* varying from 1000 to 4000. The total number of mashups requested by end-users varies from 1000 to 10000 where mashups request rate varies from 5 to 65 requests per unit time. The data sources are extracted from syndic8 [3] which is a repository for RSS and Atom feeds, the popularity distribution of data sources is also extracted from syndic8 where the number of subscriptions for a data source reflects its popularity. The number of operators per mashup varies from 10 to 40 where two of these operators are fetch operators and their data sources are selected based on data sources popularity, the rest of operators are selected randomly. Data source sizes vary from 1000 KB to 10000 KB. Our overlay topology is the Internet topology in 2008 measured by DIMES [13]. The number of nodes we use from this topology varies from 1204 to 4204.

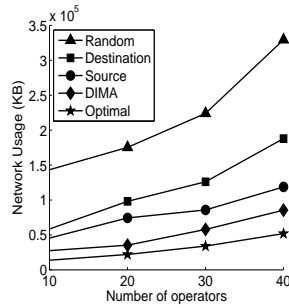
### 6.2 System Evaluation

System cost in *CoMaP* is measured based on two factors, first, the average delay per mashup resulting from deploying and executing mashups operators, second, the average network usage per mashup which is defined as the average number of bytes transferred within the network caused by executing mashups. The ‘*DIMA*’ approach is compared to ‘*Random*’, ‘*Source*’, ‘*Destination*’, and ‘*Optimal*’ approaches. In all experiments we vary one parameter and keep the others constant. Unless mentioned, the constant values for number of mashups, number of operators, mashup request rate, data source size, and number of *MPNs* are 1000 mashup, 10 operators per mashup, 25 requests per unit time, 1000 KB, and 2000 *MPNs*, respectively.

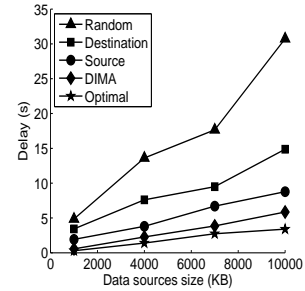
In the first experiment we vary number of operators in *CoMaP* from 10 to 40 and we measure delay and network usage for the different schemes; Figures 5



**Fig. 5.** Average delay per mashup when number of operators varies



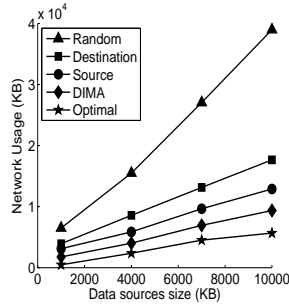
**Fig. 6.** Average network usage per mashup when number of operators varies



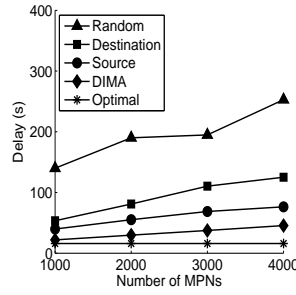
**Fig. 7.** Average delay per mashup when data source size varies

and 6 show that ‘*Random*’ scheme leads to high delays and high network usage and it is the worst among all schemes because it does not follow any kind of heuristics to deploy operators. The ‘*Source*’ and ‘*Destination*’ schemes lead to lower delay and network usage than the ‘*Random*’ deployment. The results of the ‘*Source*’ and ‘*Destination*’ schemes might vary depending on how many end-users share operators. The ‘*Optimal*’ scheme generates the lowest delay and network usage which is expected because of the exhaustive search performed by this scheme. This scheme is not practical because it requires long exhaustive search. ‘*DIMA*’ approach beats ‘*Random*’, ‘*Source*’, and ‘*Destination*’ approaches in terms of delay and network usage. This better performance is the result of a more dynamic two-stage optimization scheme that depends on distances from data sources and end-users at the same time, and it depends on operator migration which keeps *CoMaP* adapting to changes in network links delay and bandwidth and to changes in number of end-users sharing operators. Notice that ‘*DIMA*’ performance is also close to the ‘*Optimal*’ approach which proves its effectiveness. Figures 5 and 6 also show that the gap between each scheme and the ‘*Optimal*’ scheme widens as more operators are used in mashups, this occurs because as more operators are used, more delay results normally from executing those operators. This delay is minimum in ‘*DIMA*’; because it uses migration to find better deployment options while migration is not used by the other schemes causing their delay of executing operators to increase rapidly.

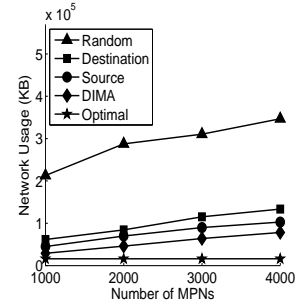
We performed another experiment where sizes of data sources varies from 1000 KB to 10000 KB, as figures 7 and 8 show, the system cost increases for all schemes as data source sizes increase, this is due to increasing computation and communication costs resulted from increasing volume of data. In the next experiment, we vary number of *MPNs* in the network from 1000 to 4000 and measure delay and network usage. The results are plotted in Figures 9 and 10. The important point to take from these two figures is that the gap between ‘*DIMA*’ scheme and ‘*Optimal*’ scheme increases because as more *MPNs* are used, the search space size increases which adds more challenge for the ‘*DIMA*’ scheme



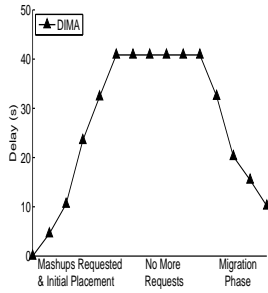
**Fig. 8.** Average network usage per mashup when data source size varies



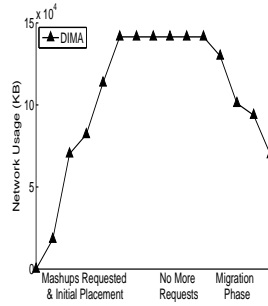
**Fig. 9.** Average delay per mashup when number of MPNs varies



**Fig. 10.** Average network usage per mashup when number of MPNs varies



**Fig. 11.** Average delay per mashup in different execution periods

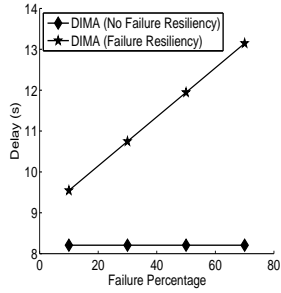


**Fig. 12.** Average network usage per mashup in different execution periods

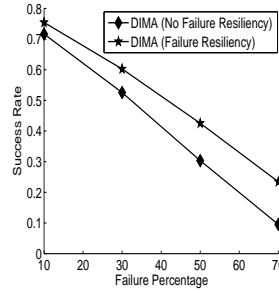
to find close to optimal deployment decisions.

We conducted an experiment to evaluate the effect of migration on system cost where we measure delay and network usage in three different periods of system execution. In the first period, mashups are requested, ‘*DIMA*’ initial operator placement is executed (stage 1), and mashups are executed. The second period continues mashup execution without further requested mashups. The third period is when ‘*DIMA*’ migration process (stage 2) starts and no further mashups are requested. As captured in Figures 11 and 12, delay and network usage increase when more mashups are requested in the first period, the system then stabilizes on the highest costs in the second period when no more mashups are requested, then delay and network usage drop significantly when the migration process starts. This cycle continues through the life time of *CoMaP*.

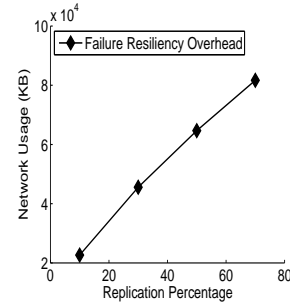
In the next experiment, we study the effect of enforcing failure resiliency in *CoMaP*. Here, the number of mashups is fixed at 10,000, replication percentage is fixed at 25 percent of the most overloaded *MPNs*. Figure 13 shows the average delay per mashup resulting from executing mashups in ‘*DIMA*’ approach in two



**Fig. 13.** Average delay per mashup in DIMA in the presence and absence of failure resiliency



**Fig. 14.** Success rate of executing mashups in DIMA in the presence and absence of failure resiliency



**Fig. 15.** Total overhead of applying failure resiliency when replication percentage is variable

cases, the first uses failure resiliency and the second does not use it. In the later case, the delay is constant because no failures are assumed in the system. In the former case, *MPNs* failure is simulated in the system based on a variable failure probability (10 percent - 70 percent) and only one mashup controller fails. We notice that applying failure resiliency increases delay. This can be explained in three points. First, end-users normally send their mashups to the mashup controller closer to them, now, when that mashup controller fails, end-users would have to send their mashups to a different farther away controller. Second, since operators are replicated on different *MPNs*, mashup execution might not go through the *MPN* that yields less delay; this happens because mashup execution is sometimes directed to different *MPNs* because the *MPNs* that yield minimum delay failed. Third, when failure probability increases, more *MPNs* that yield minimum delay fail which forces communication to be directed to other *MPNs* which do not lead to minimum delay.

In the next experiment, we measure the success rate of executing mashups by using the same parameters as the previous experiment except that failure is assumed in both cases of existence and absence of failure resiliency. Figure 14 shows that success rate decreases as failure probability increases and that is a direct effect of increasing nodes failures. The same figure shows that using failure resiliency results in higher success rate than the case where failure resiliency is absent; this is because operators are replicated which increases their availability. In the last experiment, we show the overhead of failure resiliency. In Figure 15, failure probability is set to 20 percent, one mashup controller fails, number of mashups is set to 1,000. The replication percentage varies between 10 percent and 70 percent which reflects the percentage of nodes their operators gets replicated. The figure shows the total overhead needed to maintain state of the system when applying failure resiliency. The overhead is measured in terms of network usage in kilobytes resulting from exchanged messages due to failure resiliency and replications. The state of the system includes keeping mashups information

on mashup controllers identical. It also includes the cost of communication between controllers in case one of them fails. It also includes the communication between MPNs and controllers in case of replicating operators and failed *MPNs*. We notice that the overhead increases as replication percentage increases, this happens because more operators are replicated and therefore more communication occurs between *MPNs* and controllers. The system faces this kind of overhead only when the system is initialized with replicas, during a failure, and when replicating operators. Other than these times, the system does not deal with this overhead.

The previous set of experiments show the effectiveness of the ‘*DIMA*’ scheme in reaching operator deployment decisions leading to low network delay and usage. Despite of the overhead of applying failure resiliency in *CoMaP*, using it decreases the failure probability of *CoMaP* by avoiding single point of failures.

## 7 Related Work

Since the advent of Web 2.0, several mashup platforms have been proposed in the literature [14] [16] [6] [7] [11]. Karma [14] offers a build by example approach for end-users. Marmite [16] works at the user end as a Firefox plug-in. Mash-Maker [6] is a tool for building mashups from widgets, it also enables end-users to share mashup widgets. DAMIA [7] is a data integration service dedicated for enterprise domain and situational applications. MARIO [11] is a mashup platform in which a planning algorithm for mashup execution is proposed where mashups are created using tags. None of the previous mashup platforms is distributed which limits their scalability.

Several platforms for distributed component execution have been investigated in literature [10] [1] [4] [5] [12]. A platform for distributed stream processing is proposed in [10]. Each stream is processed in several broker nodes in an overlay network; authors propose a scheme for placing stream processing operators. The cost metric we propose in *CoMaP* is more comprehensive than the metric they use. Also, authors do not consider failure issues while we do target this issue. Another operator deployment environment is proposed in [1] where deployment is based on Distributed Hash Tables (DHT) routing. However, as explained in [10], using DHT for selecting nodes where operators are deployed can lead to poor node selections. Borealis [4] is a distributed stream processing system, where operator deployment in their system does not consider network overlay changes such as changes in links delay and bandwidth. Medusa [5] is another distributed stream processing environment that deploys operators in a way to achieve load balancing. Analysis for node placement in overlay networks is investigated in [12]. However, they focus on placing machines on network infrastructure, while *CoMaP* focuses on mashup operator placement in overlay networks.

## 8 Conclusion

As one of the collaborative Web 2.0 applications, mashups are faced with a number of scalability and performance limitations. In this paper, we presented *CoMaP* – a dynamic cooperative overlay-based mashup platform. *CoMaP* incorporates several novel features. First, we presented a scalable and efficient architecture for *CoMaP* comprising of a multitude of cooperative mashup processing nodes and a set of mashup controllers. Second, we introduced a dynamic mashup distribution technique that is sensitive to the relative locations of the sources and the destinations of a mashup, and optimizes data flow within the overlay. Third, we described how we enforce failure resiliency feature in our system. Load balancing is an important feature to be applied in our system as a future work. Our experimental study demonstrated that *CoMaP* yields improved system performance and scalability.

## References

1. Ahmad, Y., Cetintemel, U., Jannotti, J., Zgolinski, A., Zdonik, S.: Network awareness in internet-scale stream processing. *IEEE Data Eng. Bulletin* 28(1), 63–69 (2005)
2. Al-Haj Hassan, O., Ramaswamy, L., Miller, J.A.: Enhancing scalability and performance of mashups through merging and operator reordering. In: *ICWS(To Appear)* (2010)
3. Barr, J., Kearney, B.: Syndic8 feeds repository. <http://www.syndic8.com> (2001)
4. Centintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryzkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the borealis stream processing engine. In: *CIDR* (2005)
5. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.: Scalable distributed stream processing. In: *CIDR* (2003)
6. Ennals, R.J., Garofalakis, M.N.: Mashmaker: mashups for the masses. In: *ACM SIGMOD*. pp. 1116–1118 (2007)
7. IBM Corp.: Damia. <http://services.alphaworks.ibm.com/damia/> (2007)
8. Intel Corp.: Mash maker. <http://mashmaker.intel.com/web/> (2007)
9. Ng, E.T.S., Zhang, H.: A network positioning system for the internet. In: *USENIX Annual Technical Conference* (2004)
10. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: *ICDE* (2006)
11. Riabov, A., Bouillet, E., Feblowitz, M., Liu, Z., Ranganathan, A.: Wishful search: interactive composition of data mashups. In: *WWW*. pp. 775–784 (2008)
12. Roy, S., Pucha, H., Zhang, Z., Hu, Y.C., Qiu, L.: Overlay node placement: Analysis, algorithms and impact on applications. In: *ICDCS*. pp. 53–53 (2007)
13. Shavitt, Y., Shir, E.: Dimes: let the internet measure itself. *ACM SIGCOMM* 35(5), 71–74 (May 2005)
14. Tuchinda, R., Szekeley, P., Knoblock, C.: Building mashups by example. In: *International Conference on Intelligent User Interfaces*. pp. 139–148 (2008)
15. Wikipedia: Web 2.0. [http://en.wikipedia.org/wiki/Web\\_2.0](http://en.wikipedia.org/wiki/Web_2.0) (2007)
16. Wong, J., Hong, J.: Making mashups with marmite: towards end-user programming for the web. In: *CHI*. pp. 1435–1444 (2007)
17. Yahoo Inc.: Yahoo pipes. <http://pipes.yahoo.com/> (2007)