Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.083861

ARTICLE

A Hybrid Mashup Platform Based on Structured and Unstructured Peer-to-Peer Networks Empowered with Genetic Algorithms

Osama Al-Haj Hassan[1,*], **Ammar Odeh**[1], **Abdullah Aref**[2] **and Ghassan Samara**[3]

[1]Department of Computer Science, Princess Sumaya University for Technology, Amman, Jordan
[2]Department of Data Science, Princess Sumaya University for Technology, Amman, Jordan
[3]Department of Computer Science, Zarqa University, Zarqa, Jordan

*Corresponding Author: Osama Al-Haj Hassan. Email: o.alhajhassan@psut.edu.jo

Received: 13 April 2026; Accepted: 28 May 2026; Published: 23 July 2026

ABSTRACT: Mashups are among the key web technologies that provide end-users with customizable and personalized tools. Most mashup platforms are based on centralized architectures or do not employ fully decentralized architectures; therefore, in this paper, we propose a decentralized architecture for mashups that combines the strengths of structured and unstructured peer-to-peer networks. For the structured part, we rely on the Chord lookup protocol, and for the unstructured part, we build groups of nodes via two flavors of network flooding, namely, sequence number flooding and reverse path flooding. Brokers in the unstructured part would be responsible for hosting and executing mashups, such that deciding which brokers should host a given mashup is determined by utilizing genetic algorithms. We compare our work against several approaches that rely on random and greedy mashup placement. We also assess our proposed approach to pure structured and pure unstructured approaches. We evaluate our system using simulations, and results show that executing mashups using the version of our scheme that relies on reverse path flooding generates at least 25% lower delays than the other approaches.

KEYWORDS: Mashup; peer to peer; Chord; genetic algorithm

1 Introduction

One of the pivotal web technologies is mashups. A mashup provides end-users with the ability to create new personalized and customized tools out of existing sources. The sources are extracted by using Application Programming Interfaces (APIs), and the extracted data is usually presented using a standard format such as HTML, XML, or JSON. Further processing is applied on the extracted data, such as merging, filtering, sorting, truncation, or any special tailored operation offered by the mashup platform. Fig. 1 depicts the operation of mashups.

Mashups are employed in several modern-day technologies such as workflow automation and orchestration [1,2] and API integration [3]. Exemplary current real-world tools are Pipes Digital [4] and IFTTT [5]. These tools are utilized for workflow creation where users deal with mashup data sources, APIs, and operations using visual drag and drop capabilities. Types of data sources the end-user can utilize are feeds, microservices, cloud-hosted services, and scraped webpages.

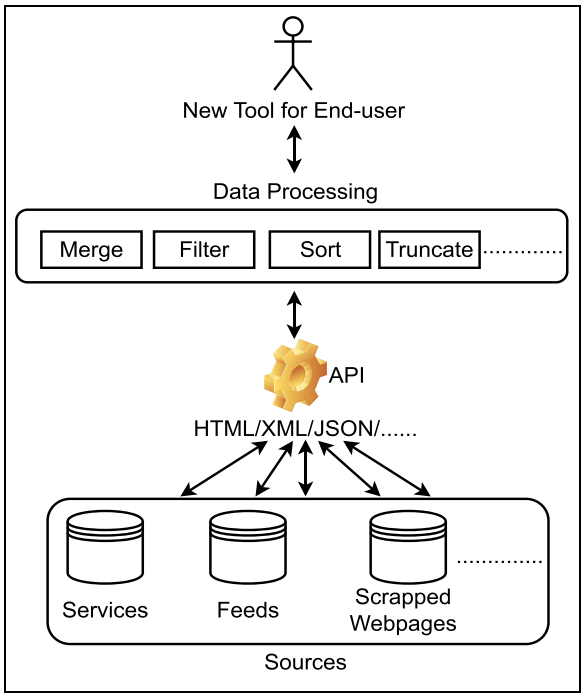

Figure 1: The basic components of mashups and the interactions between them, which show the way mashups operate.

1.1 Motivation

In its essence, mashups are personalized web services. However, there are distinct differences between mashups and web services, which impose more careful design on mashup platforms. First, a web service is created by a given organization for the use of groups of people interested in that web service, while a mashup is created by an end-user to satisfy the end-user's own needs. This personalization property of mashups implies that the potential number of mashups hosted by a mashup platform would be higher than the number of web services that exist on a web service portal. Hence, mashup platforms can incur higher request rates than web service portals.

Most of the existing mashup platforms are based on centralized architectures or follow a partially decentralized architecture, which causes a serious bottleneck scalability issue. Also, given the demands of mashup platforms that we just explained, a centralized server architecture would work against those demands. Therefore, a decentralized architecture for mashups is needed.

1.2 Research Gap

The existing mashup platforms are based on centralized architectures such as "Pipes.Digital" [4], which is currently a working mashup platform in the industry. Another work is presented in [6], which offers a mashup platform that includes interactions between several data sources, users, and servers. Although their work utilizes several servers, they still rely on the client-server interaction model, which does not represent a fully decentralized architecture, and this, in turn, raises scalability issues. Several works pertaining to mashups can be found in the literature, for example, the works in [7,8] introduce ranking techniques that rank data sources that can be used in mashups and that allow end-users to identify the most valuable data sources. Also, the works in [9,10] discuss the problem of the end-users being overwhelmed by a vast amount of data sources which can be used in mashups, and therefore, the authors propose mechanisms to alleviate this problem and make the end-user's life easier in choosing appropriate data sources. Privacy aspects of

mashups are discussed in [11]. The previous works are examples of the state of the art that we mention in the related work section, and the point we make here is that the previous works rely on centralized architecture, which causes them to be prone to failures.

Another way to search for data employs Large Language Models (LLMs). LLMs rely on artificial intelligence models that are trained on large amounts of data, and this requires an expensive infrastructure. In contrast, mashups do not require any training on data, and they are cheaper to deploy.

Based on that, we found that a decentralized architecture for mashup platforms is needed, and to the best of our knowledge, current mashup platforms are based either on a centralized server or a partially decentralized architecture. Accordingly, in this work, we are trying to answer the following research questions. First, **RQ1:** Can we build a decentralized architecture for mashups that facilitates mashup hosting and execution in a decentralized manner? **RQ2:** Is it possible to provide efficient message routing in such an architecture?

1.3 Research Objective

In this paper, we aim to overcome the shortcomings of the centralized mashup architectures employed in works such as [4,6] by proposing a decentralized architecture for mashup platforms. The proposed decentralized architecture is based on peer-to-peer architectures. Employing peer-to-peer architectures gets rid of the problems of the ordinary client-server model. However, there are two classes of peer-to-peer architectures, namely, structured and unstructured peer-to-peer architectures. Each of them has its own merits and shortcomings. Therefore, we investigate the use of a hybrid peer-to-peer architecture that combines the best of the structured and unstructured peer-to-peer architectures. To that end, our work embodies the following contributions.

- Proposing a decentralized architecture for mashups based on a hybrid model of structured and unstructured peer-to-peer networks.
- Utilizing genetic algorithms for the purpose of deciding which network nodes should host mashups.
- Providing message routing through the proposed architecture in a way that facilitates mashup execution.

To the best of our knowledge, a fully decentralized architecture for mashup execution based on both structured and unstructured peer-to-peer networks does not exist, and providing one with the previous contributions is a step toward more scalable mashup platforms.

The rest of the paper is organized as follows. Section 2 discusses related works in the field of mashups. Then, Section 3 proposes our methodology which includes explaining how we represent mashups in our system in Section 3.1, introducing the system architecture in Section 3.2, presenting the structured part of our system in Section 3.3, describing the unstructured part of our system in Section 3.4, exploring how to utilize genetic algorithms to map mashups to the brokers responsible of executing them in Section 3.5, and portraying how message routing is handled in the system in Section 3.6. Then, we discuss security and reliability aspects of our system in Section 4. This is followed by Section 5 that sheds light on system complexity. After that, we evaluate our system in Section 6. Finally, we conclude this work in Section 7.

2 Related Work

In this section, we discuss existing works in the literature regarding the areas of mashups and peer-to-peer networks.

2.1 Mashups

Several works related to mashups exist in the literature. Authors in [12] utilize ant colony optimization to find the shortest distance between several destinations the end-user wants to visit. Authors build a mashup that constitutes locations and paths on maps. In their work, a mashup is a result generated by a centralized server service, and they never discuss any information delivery scheme in their work. The end-user is overwhelmed by the sheer amount of sources on the Web, and therefore, an approach based on genetic algorithms is proposed in [9] to assist end-users in selecting data sources for their mashups, and the approach achieves this by minimizing the cost of executing mashups. The execution of the utilized genetic algorithm occurs on a centralized entity. The enormous amount of data sources impedes the work of end-users in picking the ones they want to use in their mashups. Accordingly, works such as [3,13,14] offer techniques to solve this problem. So, the work in [14] detects discrete attributes and semantic relationships of data sources and resembles them in a graph, and this graph is annotated with user preferences, which aids in recommending data sources to end-users. In [3], the authors tackle the issue of the lack of presence of domain knowledge for end-users, which hampers their ability to build mashups, and therefore, the authors study the semantics of the contents of data sources and recommend APIs of sources for end-users. In [13], an issue related to the popularity of data sources is discussed, where the popularity of certain data sources creates some kind of bias for end-users to keep using them in their mashups, and that impairs discovering new and possibly valuable data sources. The works in [3,13,14] rely on centralized architectures, which cause a single point of failure problem. Recommendation techniques that can be used to recommend APIs and services for end-users to use in mashups are discussed in a survey in [15]. A point of view of security is introduced in [16] where authors mention that executing mashups requires extracting data from data sources, and if those data sources happen to carry malicious code, then this code would end up being executed on the end-user's machine, which makes it vulnerable to security attacks. The effect of programming skills and their effect on the ability of end-users to build useful mashups are investigated in [17].

The previous works pertain to the domain of mashups; however, they do not adopt decentralized mashup architectures.

2.2 Peer to Peer Networks

A key player in network technology is peer-to-peer networks, and we mention several works in this domain in the section. The work in [18] employs Honey Bee Optimization to optimize paths between nodes in unstructured peer-to-peer networks. The work also optimizes the throughput of the network; however, it suffers from inefficient routing in unstructured peer-to-peer networks. Authors in [19] propose an approach to maintain the ability of unstructured peer-to-peer networks to expand even in the presence of high rates of churn, but their work also relies only on unstructured peer-to-peer networks. The work in [20] shows how to build unstructured peer-to-peer networks that can function as multi-hubs. Their overlay maintains the use of randomness of joining the network while keeping the shape of a star network, which results in a robust and low-diameter overlay; however, as in the previous works, their work depends solely on unstructured peer-to-peer networks, which causes routing deficiencies. Authors in [21] state that the performance of federated learning models suffers when applied on a centralized server architecture; therefore, they propose building a federated learning model on top of a Chord structured peer-to-peer architecture. Authors also utilize aggregation to mitigate cases where there are a variety of slow and fast nodes. Their work harvests the benefits of structured peer-to-peer networks, but at the same time, relying exclusively on structured peer-to-peer networks adds to system complexity. In [22], authors tackle the problem of inefficient cloud analytics for health care systems that are based on a centralized server and lack context awareness. So, they propose a hybrid architecture based on peer-to-peer networks. The structured part of their architecture is based on

an RC structure, which allows their system to establish groups of nodes based on the specific interests of users. This work reaps the strength of combining both structured and unstructured peer-to-peer networks; however, it does not target mashup platforms. The work in [23] proposes a federated learning system for edge computing such that the system utilizes a publish/subscribe architecture built on top of a hybrid peer-to-peer network. The structured part of their work relies on multiple rings built using Chord. Similar to the previous work, this work utilizes both structured and unstructured peer-to-peer topologies, but it does not focus on mashup platforms.

We summarize the features of the existing works in literature and our work in Table 1 such that the columns in the table are interpreted as follows:

- Mashup Centric: Indicates if the work targets mashup platforms.
- Architecture: States that the employed architecture is a centralized server (CS) or a distributed network (DN).
- Optimizing Delay: Mentions whether the work optimizes network delay.
- Meta Heuristic Optimization: Declares if any meta-heuristic optimization is utilized in the work.
- P2P: Lists the type of peer-to-peer (P2P) topology used in the work if any.

Our work is the only one among the listed works that satisfies the following features. First, being dedicated to mashups. Second, being based on decentralized architecture. Third, relying on a hybrid peer-to-peer architecture. Fourth, utilizing an evolutionary approach, such as genetic algorithms, to optimize mashup execution delays.

Table 1: Comparison of our work and the state of the art.

Work Citation	Mashup Centric	Architecture	Optimizing Delay	Meta-Heuristic Optimization	P2P
[6,3,13,14,17]	Yes	CS	No	None	None
[9]	Yes	CS	Yes	Genetic Algorithm	None
[12]	Yes	CS	Yes	ant colony	None
[18]	No	DN	Yes	Honey Bee Optimization	Unstructured
[19,20]	No	DN	No	None	Unstructured
[21]	No	DN	Yes	None	Structured
[22]	No	DN	Yes	None	Hybrid
[23]	No	DN	Yes	Bandit-based Optimization	Hybrid
Our work	Yes	DN	Yes	Genetic Algorithm	Hybrid

3 Methodology

In this section, we introduce our mashup distributed architecture and describe its details.

3.1 Mashup Representation

Each mashup has a corresponding string representation, which plays an important role later when we map each mashup to a broker in the system. Therefore, it is important to clarify how we build the string representation of a given mashup. For that matter, we look into the mashup depicted in Fig. 2. The mashup is intended to collect information related to the sports and entrepreneurship news of the athlete "Lebron James". The mashup extracts data from the New York Times ProBasketball sports feed (https://rss.nytimes.com/services/xml/rss/nyt/ProBasketball.xml), then the extracted data is filtered based on "Desc" containing "hall

of fame". In addition, the mashup extracts data from the entrepreneur.com feed (https://www.entrepreneur.com/latest.rss). Then, the extracted data from both data sources is merged and then filtered based on "title contains Lebron James". After that, the data is sorted in descending order based on publication date and truncated to keep the latest 5 items. In this example, we have used feeds as the type for data sources; however, the concepts in this work are not restricted to a given data source type.

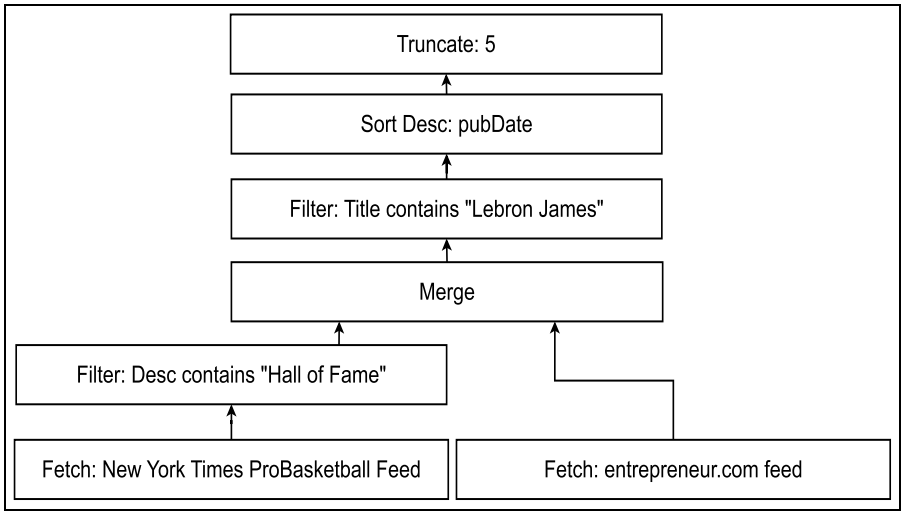

Figure 2: Simple example of a mashup.

Each mashup operator has an equivalent string representation, and since a mashup consists of several mashup operators, the concatenation of the representation of operators results in the final mashup representation. To further explain this, we look at the string representation of the mashup in Fig. 2, the first filter operator "New York Times ProBasketball sports feed" is represented by 40, which is a chosen ID for the data source. Similarly, the second fetch operator "entrepreneur.com feed" is represented as 42. What comes next is the filter operator "Filter: Desc contains hall of fame" which is represented by the ID of the filter operator (15), followed by the filter predicate, and therefore its basic representation becomes "15:Desc contains hall of fame". Now, we can represent the filtered data from data source 40 by concatenating the string representations of both operators, using the # symbol as a separator between them. Accordingly, the representation of the filtered data becomes "40#15:Desc contains hall of fame". This is followed by the join operator, which has a representation that starts with "SU" indicating the start of a join block, and "EU" which refers to the end of the join block, and "MU" which stands in between the two merged sources. Accordingly, the representation of the merge operator becomes "SU:40#15:Desc contains hall of fame: MU:42:EU" which represents joining the filtered result of data source 40 with the fetched data of data source 42. The merged data is then filtered based on title containing Lebron James, and it is represented in a similar way to the first filter operator. However, the data to be filtered is the data resulting from the join operator. Therefore, we follow the same concatenation process to append the filter operator representation to the join operator representation, which leads to the result "SU:40#15:Desc contains hall of fame:MU:42:EU#15:title contains Lebron James". The sort operator "Sort: pubDate desc" comes next, and it is represented by the ID of the sort operator followed by the attribute upon which data is sorted and the type of the sort operation, so its representation becomes "09:desc on pubDate". Accordingly, the representation of the mashup until this point is "SU:40#15:Desc contains hall of fame:MU:42:EU#15:title contains Lebron James#09:desc on pubDate". Finally, the result is truncated such that 5 items are kept from the result, and this is represented with the ID of the truncate operator (17) followed by the required number of items to keep; hence, its representation becomes "17:5". Consequently, the final

representation of the mashup becomes "SU:40#15:Desc contains hall of fame:MU:42:EU#15:title contains Lebron James#09:desc on pubDate#17:5".

3.2 System Architecture

Structured and unstructured peer-to-peer networks are among the popular architectures used for distributed systems. This is due to the lack of central control that usually exists in ordinary client/server models. This makes peer-to-peer architectures excellent for graceful scalability. However, each of these two architectures has its own advantages and disadvantages [24]. On the one hand, structured peer-to-peer architectures impose a certain organization on the way peers are connected to each other, which makes them organized and scalable. In addition, this organized connectivity facilitates efficient search and routing through the structured topology, which is usually performed using distributed hash tables (DHTs). However, the imposed organized connectivity introduces a certain degree of complexity to structured topologies. On the other hand, unstructured peer-to-peer networks do not impose a certain way for peers to be connected to each other, and therefore, one of the main advantages of them is their simplicity and convenience. This also allows them to be more dynamic due to loose constraints on node connectivity. However, this simplicity comes at the expense of poor search efficiency, which causes high delays. To this end, we aim to combine both structured and unstructured topologies in our system architecture, which allows us to harvest the best of the two topologies.

The system architecture and network overlay are depicted in Fig. 3. The network overlay consists of several end-users, brokers, and data sources. End-users design mashups, and this is why they have the component "Mashup Designer". which normally provides drag-and-drop capabilities to build mashups. They also have a "Mashup Storage" component to store the mashups they design. In addition, they have a "Broker Interface" component, which allows them to request the execution of mashups from brokers in the structured part of the topology.

Brokers form a network, and they exist in both the structured and unstructured parts of our architecture. However, a broker's role differs based on belonging to the structured or the unstructured parts. A broker in the structured part takes part in several tasks as follows. First, it receives the end-user's request to execute a given mashup. Second, upon receiving the end-user's request, the broker would check if the mashup to be executed is a new one in the system or a previously existing one. If it is a previously existing one, then the broker would forward the request to a broker in the unstructured part it governs. However, if the mashup is a new one, this would engage the "Genetic Optimization" component to decide which of the brokers in the unstructured part would be responsible for hosting and executing the mashup. Third, the broker utilizes the "Forming Groups" component to build the unstructured group of brokers that it will moderate. The "Genetic Optimization" component requires accessing mashups and therefore the broker holds a "Mashup Storage" component.

For the brokers in the unstructured part, when a broker receives a mashup execution request, then it executes the mashup via the "Mashup Execution Engine" component and sends the result to the responsible broker in the structured topology, which in turn hands the result to the end-user. The "Mashup Execution Engine" requires the need of the "Mashup Storage" component to host mashups and access them when they need to be executed.

Data sources resemble sources in the Web, such as feeds, and those will be contacted when a given mashup is executed. When data is extracted from these sources, it is usually returned in a standard format such as HTML, XML, or JSON.

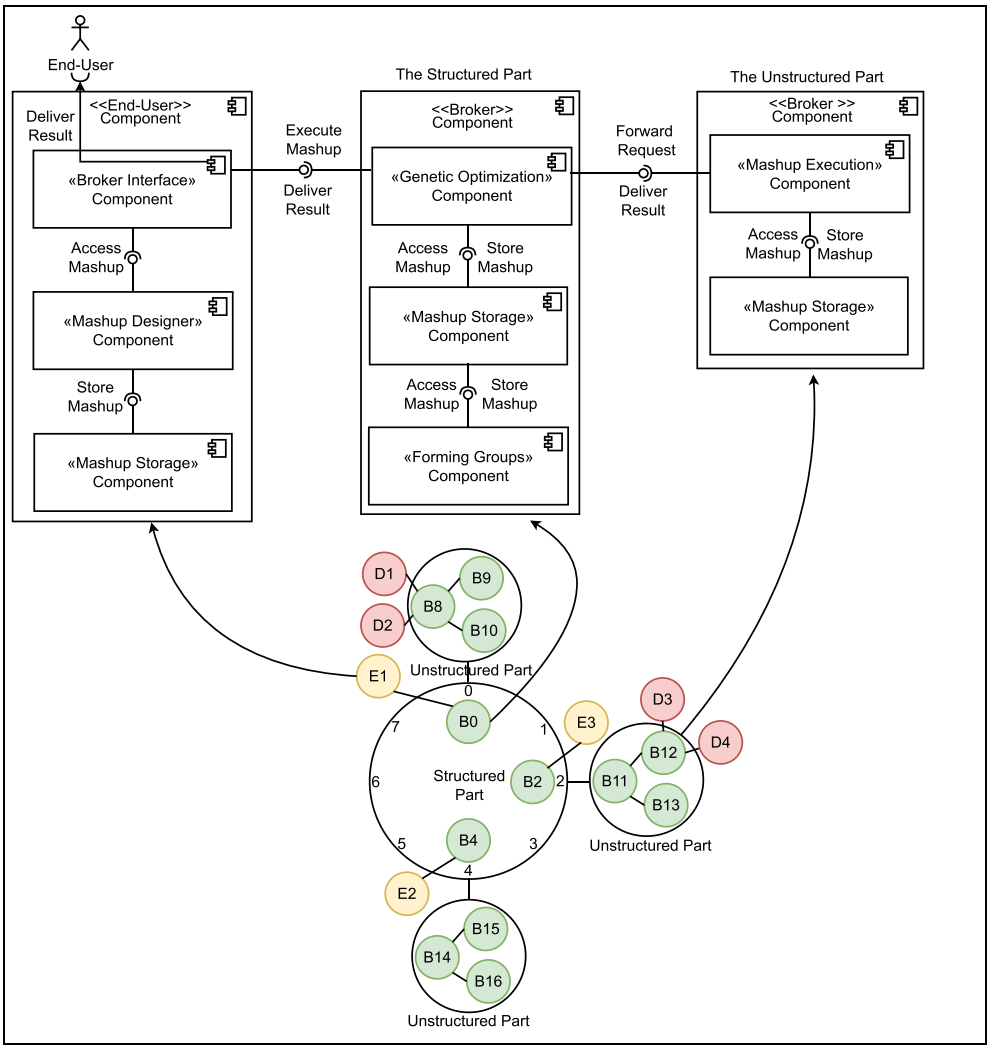

Figure 3: System architecture showing the interactions between the several components existing on the end-user machine (denoted by "E"), the broker in the structured part of the architecture, and the broker in the unstructured part of the architecture, where brokers are denoted by "B". Data sources (denoted by "D") are needed for mashup execution.

3.3 The Structured Part

Here, we describe the details related to the structured part. For the structured part, we follow the topology of Chord [25], which relies on the concept of consistent hashing and distributed hash tables.

Accordingly, for the structured part, brokers are organized using a virtual ring topology where it is possible to place "n" brokers on a given ring such that "n" is a power of 2. Even when all the "n" places are taken on the ring, the ring can expand to make room for additional brokers. The way a broker is placed on the ring topology is governed by applying a hash function on the broker's IP address. For that purpose, the SHA-1 hash function is utilized based on Eq. (1). Let us explain that using an example. As shown in Fig. 3, the ring allows for 8 brokers numbered from 0 to 7, and 3 of those 7 places are taken by brokers B_0, B_2, and B_4. So, when a given broker is to be added to the ring, its IP address needs to be translated to one of the available identifiers from 0 to 7. Passing the IP address of the broker to the SHA-1 function results in a hexadecimal number output, then the hexadecimal number is converted to a decimal number. However, the resulting decimal number might not fall within the allowed 8 identifiers (0 to 7). To solve that mathematically, we

divide the resulting decimal number by 2^m, where m is the number of binary digits needed to represent the "n" possible identifiers. In this case, m = 3 because $2^3 = 8$, then the remainder of the division becomes the broker identifier on the ring.

$$ID = ToDecimal(SHA1(R)) \ mod \ 2^m \tag{1}$$

In our platform, brokers host mashups, and specifically, we assign this task to the brokers in the unstructured part of the architecture. Therefore, we need to find a way that uses the representation of a given mashup to map it to a broker in the structured part. This paves the way to the unstructured brokers network, which the broker in the structured part is responsible for, and then the mashup would be hosted by one of the brokers in the unstructured part. Hence, the first step for us is to map the mashup representation to a broker in the structured part of the network, and to achieve that, we follow the way employed in Chord. Based on that, the representation of a given mashup is also passed to Eq. (1) and the result would be a decimal number (MashID) such that the broker on the structured part that is mapped to this mashup is successor(MashID) where the successor function is a simple function that returns the broker on the ring topology which has the identifier that is equal to MashID or the closest to MashID clockwise. For example, successor(0) is 0 because there is a broker B_0 at position 0 on the ring; however, successor(1) = 2 because there is no broker occupying position 1, and therefore, we look for the next broker clockwise, and that would be broker B_2. Let us discuss another example in the context of mashups. Suppose a given mashup results in MashID = 5. Now, looking back at Fig. 3, the broker to which the mashup is mapped is the one resulting from successor(5). We can see that the place with ID 5 is not occupied by a broker yet, so we move clockwise to find the closest broker, which is broker B_0 in this case. So, the previous mashup is mapped to broker B_0. Now broker B_0 would be responsible for finding a broker within the unstructured brokers network it is responsible for and that broker would be the broker on which the mashup will be hosted and therefore it will be responsible for executing that mashup in the future.

One thing to mention here is that searching the ring in this sequential manner is costly and accordingly Chord employs the concept of finger tables which aids in a more efficient process in finding successor brokers and it is also used in routing through the ring topology, and this way a smaller number of steps would be needed to find the broker to which a given mashup is mapped. This is realized using the distributed hash table concept employed in Chord. So, each broker in the structured part keeps a finger table such that each entry of the finger table is of the format <BrokerID, [Interval Start, Interval End), SuccessorBrokerID>, which is interpreted as follows. The broker with "BrokerID" is responsible for keeping mashups that have IDs that fall within the interval that starts with "Interval Start" inclusively and ends with "Interval End" exclusively, and the successor for the broker with "BrokerID" is the broker with "SuccessorBrokerID". In Fig. 4, we show the structured part of our architecture where each broker in the structured part has its own finger table. Looking at the previous figure, let us assume that broker B_2 is trying to find out where the mashup with ID 5 is hosted. The broker searches its finger table for an interval that includes the MashID 5. In this case, the second entry has an interval [4,6), and it includes MashID 5. The corresponding broker ID is 4, and based on the fact that the broker hosting a given MashID should have an ID equal to MashID or the closest broker with ID greater than MashID, broker B_4 precedes identifier 5, so we go for successor(4), which is 4 based on the entry in the finger table. Reaching broker B_4, the same process is repeated. Identifier 5 is found in the first entry interval [5,6), and the broker ID is 5, and the successor(5) is 0. Here, the process stops because 0 is the successor of 5 going clockwise in the ring. Consequently, broker B_2 now would contact broker B_0 for any future requests to execute the mashup with MashID 5. For simplicity reasons, we refrain from explaining how finger tables are built since this is Chord-specific information and it is not needed for explaining the concepts in our work; what is required is how the finger tables are utilized for searching and routing, which we already explained.

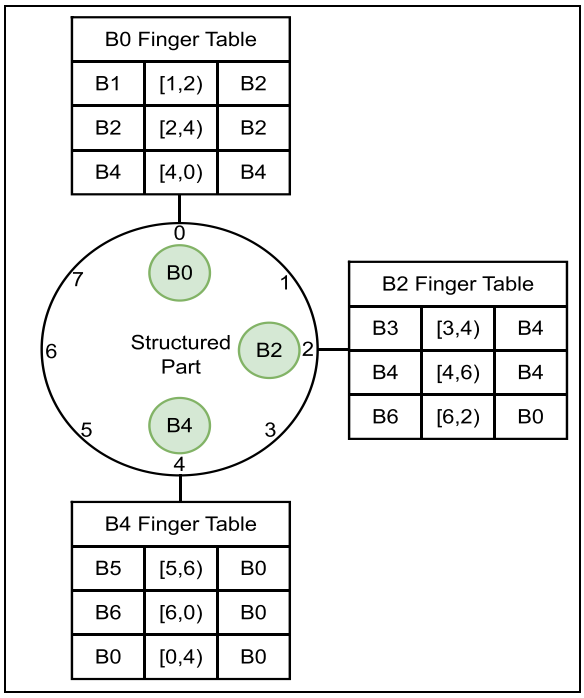

Figure 4: Showing the structured part of the system with finger tables for brokers on the Chord ring.

3.4 The Unstructured Part

Out of the brokers set in the system, several of them would join the structured Chord ring, and the rest would join the unstructured topology pertaining to one of the brokers in the structured topology. So, how can we make the distinction of which broker joins the structured or the unstructured parts of the topology? To achieve that, we form groups out of the brokers set such that one member of each group (Group Head) is the representative of that group on the structured Chord ring, and the rest of the brokers in each group belong to the unstructured part, which is under the control of the group head.

The way we form groups is depicted in Algorithm 1 and described as follows. First, the system administrator is responsible for deciding the desired number of groups (“.” number of groups), then the administrator picks one broker (B_s) randomly to be a group head for one of the groups (line 1) and then launches “k-1” random walkers to select the remaining “k-1” group heads (lines 2–6). Each random walker would travel starting from B_s as deep as it can through the network, such that the random walker behaves as follows. The random walker starts from broker B_s and randomly picks a broker from its direct neighbors, and the process is repeated until either an inevitable loop is encountered (line 4) or the random walker has already reached the allowed number of hops to travel through the network (line 5). The second condition is important because it forbids the random walker from traveling too long in the network, so each random walker is given a Time to Live (TTL) value such that the TTL value is decremented each time the random walker moves one hop ahead, and when the TTL value expires (i.e., reaches zero), then the random walker stops. The broker at which the random walker stops is chosen as another group head, regardless of the random walker being stopped due to an unavoidable loop or due to TTL expiration. The way a random walker knows if it hits an inevitable loop is by utilizing a list of previously visited brokers, which it keeps and updates through its way. After all the random walkers finish their work, the result is “k” group heads, and they will join the structured Chord ring using the same approach explained in Section 3.3.

Algorithm 1: Description of choosing group heads and building their groups

Input: -B_s: A broker chosen by the system administrator
Output: -$groupHeads$: A list of group heads
-Each broker belongs to a group head

```
1  groupHeads [0] ← B_s;
2  for i ← 2 to k do
3      groupHeads[i] ← randomWalker(defaultTTL);
4      COMMENT: Random walker stops if it faces an inevitable loop;
5      COMMENT: Random walker stops if its TTL expires;
6  end
7  for i ← 1 to k do
8      COMMENT: This is executed by each group head i in parallel;
9      Let fList represent the List of brokers from which flooding is performed;
10     fList ← groupHeads[i];
11     while (~ fList.empty) do
12         Let B_c represent the broker from which flooding continues;
13         B_c ← fList [0];
14         adjList(B_c) ← Adjacency list of B_c;
15         for j ← 1 to size(adjList(B_c)) do
16             Nbr ← adjList(B_c)[j];
17             if (Nbr.groupHead = –1) then
18                 Nbr.groupHead ← i;
19                 Add Nbr to fList;
20             end
21         end
22         Remove fList [0] from fList;
23     end
24 end
```

Now, each of the group heads is responsible for forming a group of brokers that are connected in an unstructured peer-to-peer overlay. This is a process that each group head starts independently on its own and therefore makes use of parallel execution. Each group head starts a flooding process that starts from the group head and conditionally spreads through the network (lines 7–24). To execute the flooding process, we utilize a list $fList$ to contain the brokers from which the flooding process continues (line 9), and this list initially contains the group head from which flooding would start (line 10). The flooding process keeps going until the $fList$ is empty, which indicates that there are no more brokers for the flooding process to continue from them (line 11). In each iteration of this loop, we start from the broker sitting on top of the $fList$, and we call it B_c (lines 12,13), and to reach the direct neighbors of B_c, we employ a list $adjList(B_c)$ (line 14). The broker B_c communicates with its direct neighbors (lines 15–21), and each of the neighbors Nbr (line 16) would join the group that belongs to the group head, supposing that it did not previously belong to another group (lines 17–20). Also, Nbr is added to the $fList$ so that it would be the next broker from which the flooding process continues (line 19). In line 22, B_c is removed from the $fList$ because the flooding process is done with this broker and will move on to other brokers from the list. The same process is repeated, starting from each of the direct neighbors. The process stops when all the contacted direct neighbors belong to another group. When this process is completed, all brokers in the system will be members of the unstructured topology of one of the group heads in the structured topology. We mention here that although the flooding process is known to

be a time-consuming operation, in this case, it is not expensive, and this is due to the ability of group heads to work in parallel. It is worth mentioning that during the flooding process, each travelling message keeps a list of previously visited brokers, which is used so that a message can memorize its way back to the group head. This way, when a broker joins the group of a given group head, then the broker can report back to the group head to inform it that it has become part of its group. Another use of this is the fact that a group head now knows the path that should be followed to reach a given broker within its unstructured group.

The previous way of forming groups depends on network flooding, as previously explained. We provide two flavors of the previous algorithm depending on a variation of the type of the flooding process, namely, sequence number flooding and reverse path flooding. Remember that a given group head executes a flooding process so that it finds brokers to join its group. However, the flooding process naturally involves that a given node would be reached several times from different paths. So, the previous two types of flooding determine what happens when a given broker receives a message from two different paths.

- **Sequence number flooding:** In this type of flooding, each message is associated with a sequence number, which is utilized to identify messages originating from the same source. Let us assume a simple scenario to make the concept clear. Suppose a given group head B_c started the flooding process and sent a message to its direct neighbors B_1 and B_3 such that the two sent messages have the same sequence number. Broker B_1 also forwards the message to its neighbor B_3. This means that broker B_3 would receive two messages from the broker B_c, the first one through the path "B_c -> B_3" and the second one through the path "B_c -> B_1 -> B_3". Let us assume that the message coming via path "B_c -> B_3" is received first, and the cost of that path is 8. Now, broker B_3 would report back to B_c that B_c can reach B_3 with a cost of 8, and B_3 joins the group of B_c. In addition, B_3 proceeds with the flooding process by forwarding the first received message to its neighbors. Next, B_3 receives the other message coming through the path "B_c -> B_1 -> B_3" with a cost of 5. Hence, the sequence number of the second received message is identical to the first one, then broker B_3 would not forward the second received message as part of the flooding process.
- **Reverse path flooding:** This type of flooding does not recognize the sequence number of messages and acts differently. Picking up from the same previous scenario, when broker B_3 receives the second message via path "B_c -> B_1 -> B_3" with a cost of 5, then broker B_3 checks its records and notices that it was previously reached by broker B_c with a cost of 8. Since the cost of the second received message is lower than the first one, then B_3 reports back again to B_c that it can be reached with a cost of 5 via the path "B_c -> B_1 -> B_3". Further, since the new path "B_c -> B_1 -> B_3" resulted in a lower cost, then B_3 proceeds with the flooding process and forwards the second message to its neighbors. However, if subsequent messages do not yield a lower cost, then the broker does not need to report back any new cost and does not need to forward those messages anymore.

3.5 Mashup Hosting

In this Section, we discuss how to choose a broker that is responsible for hosting/executing a given mashup. We explained in Section 3.3 that a mashup representation is translated into a mashup identifier (MashID). Then, the MashID is mapped to one of the group heads in the structured Chord ring. However, the group heads are not responsible for executing mashups; instead, they are responsible for guiding the mashup execution request to one of the brokers in their unstructured groups. Accordingly, when a given group head receives a request to execute a given mashup, it would need to instruct one of the brokers in its unstructured group to host the mashup. Nevertheless, the unstructured group of a given group head contains many brokers, and the decision of which broker of them would host a given mashup has implications that affect the delay in the system. Hence, it is the responsibility of the group head to decide which broker within its group would host a given mashup based on generating minimal delays in the system, and this is known

to be an NP-complete problem. Evolutionary techniques can be used to solve this problem, and they rely on randomized search mechanisms. In our work, we adopt genetic algorithms to decide where each mashup is hosted in the unstructured groups.

For a given group head, when it receives a mashup execution request, it would need to map the mashup to a broker in its unstructured group. This is called a solution, and it is called a chromosome in genetic algorithms vocabulary. Therefore, a chromosome looks like what is represented in Table 2, where a binary solution is represented as a one-dimensional array comprising mapping of mashups to brokers in the unstructured group. For example, the figure shows that $Mash_1$ is hosted at broker B_1, $Mash_2$ is hosted at broker B_3, and $Mash_3$ is hosted at broker B_4.

Table 2: The structure of the chromosome (solution) showing the brokers that are responsible for hosting/executing mashups.

$Mash_1$				$Mash_2$				$Mash_3$			
B_1	B_2	B_3	B_4	B_1	B_2	B_3	B_4	B_1	B_2	B_3	B_4
1	0	0	0	0	0	1	0	0	0	0	1

Each solution (chromosome) has a given value (Fitness Value) that resembles the quality of the solution. In our case, the quality is related to generating minimal delays. Accordingly, the fitness value of a given solution can be measured based on Eq. (2), where the delay is measured according to the delay resulting from extracting the data sources involved in mashups. The symbols used in the equation are as follows. $Ex(Mash_m, B_n)$ is a simple function that returns 1 if mashup $Mash_m$ resides at broker B_n and it return 0 otherwise. Similarly, $Ex(D_q, Mash_m)$ return 1 if data source D_q is used in mashup $Mash_m$, otherwise 0 is returned. The size of data extracted from data source D_q is represented as $SZ(D_q)$. $BW(D_q, B_n)$ returns the minimum bandwidth along the path from data source D_q to broker B_n. Accordingly, the optimization goal of the genetic algorithm becomes finding a solution that minimizes the delay. The optimization process needs to respect a constraint that restricts the number of copies of a given mashup to a certain number. This constraint is also stated in Eq. (2), and it is explained as follows. The default value that we use for the number of copies of a given mashup in the network is 1, and since the function $Ex(Mash_m, B_n)$ returns 1 if mashup $Mash_m$ is hosted at broker B_n, then the total summation of that value for a given mashup on all brokers should be 1. Although we deal with hosting a single copy of each mashup in the system, the constraint can be changed accordingly to allow the genetic algorithm to host more than one copy of a given mashup in the network.

$$\begin{aligned} D &= \sum_{n=1}^{N} \sum_{m=1}^{M} \sum_{q=1}^{Q} Ex(Mash_m, B_n) \times Ex(D_q, Mash_m) \times \frac{SZ(D_q)}{BW(h(D_q), h(B_n)))}, \\ &\ni \left(\sum_{n=1}^{N} Ex(Mash_m, B_n) \right) = 1 \end{aligned} \tag{2}$$

The genetic algorithm is shown in Algorithm 2, where it starts by creating an initial population of solutions such that each created solution is similar to the one in Table 2. The default population size we use is 100, such that the initial population is initialized using a standard random initialization (line 1). Then, several parameters are set for the genetic algorithm (lines 2–5). The genetic algorithm optimizes solutions by going through several iterations, and the number of iterations is set in line 2 (default is 1000). Also, the type of selection operator is set to tournament selection (line 3), where the default tournament size is 2. This selective pressure is considered moderate because two solutions are chosen at random, which leaves

a chance for weaker solutions (with high delays) to be selected, and that helps to maintain the population diversity and prevents premature convergence. In addition, the crossover and mutation operators are set to single-point crossover and bit flip mutation, respectively, where the probability that each operator takes place is determined in lines (4, 5) such that the default probability for both operators is 0.8. What happens in each iteration is illustrated in lines (6–13) such that "T" represents the number of iterations (line 6). In each iteration, tournament selection is used to select a parent list consisting of two parents (line 7), which will undergo a mating process. The mating process is performed using single-point crossover (line 8), where the probability of executing the crossover is previously set at line 4. The result of the crossover is two children, and a bit flip mutation is applied on the two children (line 9) with a probability set at line 5. Next, the delay (fitness value) resulting from each child is evaluated in lines 10 and 11. In line 12, the two children will replace two solutions with the worst fitness value in the population. This process is repeated until the number of iterations expires or the convergence condition is met. The default convergence condition we adopt is when the minimal achieved delay so far never changes for a consecutive 10% of the iterations. At the end of the optimization process, the best solution of the population is used as the result (line 14).

Algorithm 2: Description of the mashup hosting via genetic algorithm

Input: $Mash_c$, the mashup to host
Output: B_e, the broker responsible for hosting and executing $Mash_c$

```
1  pop ← initializePopulation();
2  T ← number of iterations;
3  sel ← tournamentSelection;
4  cross ← singlePointCrossoverProbability;
5  mut ← bitFlipMutationProbability;
6  for i ← 1 to T do
7      parentList ← selection(pop, sel);
8      childList ← crossover(parentList [0], parentList [1], cross);
9      childList ← mutation(childList [0], childList [1], mut);
10     evaluateFitness(childList [0]);
11     evaluateFitness(childList [1]);
12     replacement(pop, childList);
13 end
14 B_e ← minFitness(pop);
```

Deciding which brokers should host a given mashup can be achieved using other ways. Here, we consider two approaches, namely, the greedy approach and the random approach.

- **Greedy Approach:** When the group head broker receives a request to host a given mashup, it starts a greedy search to find a broker to host the mashup. The group head sends a request to its direct neighbors, asking them for the cost of hosting the mashup. Each of them computes the cost based on Eq. (2), which we explained before. All the direct neighbors report back the cost to the group head. The group head picks the neighbor that resulted in the lowest cost and sends a request to that neighbor to repeat the same greedy process, starting from that neighbor to find a lower cost than the one currently achieved. The process continues until all of the direct neighbors fail to find a lower cost than the current cost.
- **Random Approach:** When a request to host a given mashup is received by the group head broker, then the broker randomly picks one of the brokers in its group to host the mashup.

3.6 Message Routing

In this Section, we review how message routing takes place, starting from requesting mashup execution by an end-user and ending with delivering results to the end-user.

As shown in Fig. 5, when an end-user asks for the execution of a given mashup, the request is received by the broker (Group Head) in the structured part to which the end-user is directly connected. Now, if the group head sees that it did not receive this mashup before, then it starts the mashup hosting process described in Section 3.5 to decide which broker in its unstructured topology will be responsible for hosting/executing the mashup. However, if the mashup already exists in the system, then the group head broker checks its records to find the broker in its unstructured topology that is responsible for executing the mashup. Recall that an outcome of the process of forming groups was that the group head knows the paths that should be followed to reach the brokers in its unstructured group and vice versa. So, the group head forwards the request to the broker, which in turn executes the mashup and delivers its result back to the group head. Then, the group head emits the result to the end-user.

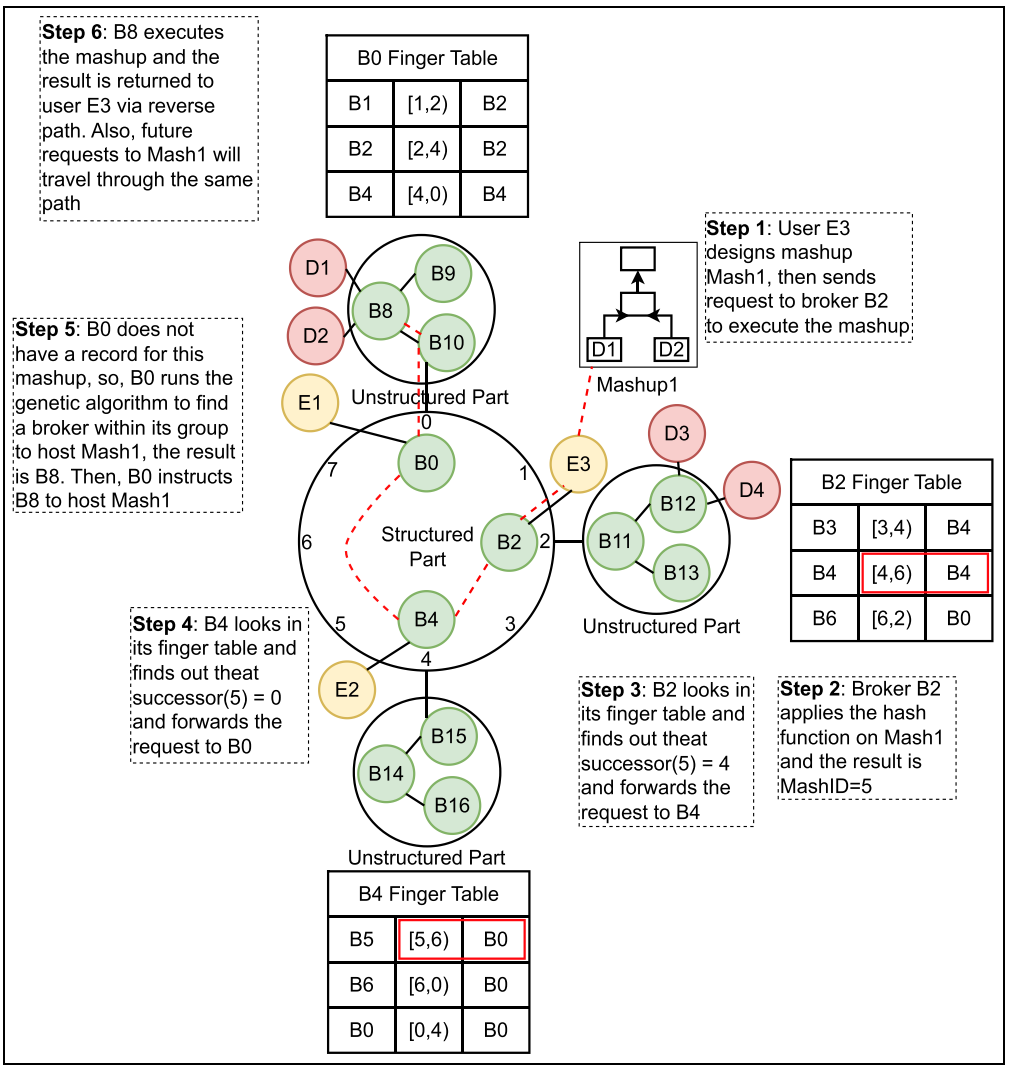

Figure 5: An illustration that shows how a user's request to execute a mashup is routed from the end-user, going through the structured part and then the unstructured part to reach a broker that hosts and executes the mashup.

4 System Security and Reliability

Security and reliability are key issues in networking systems. In this section, we discuss measures to enhance our system security and propose ways to improve its reliability against node failure and churn.

Security of mashup platforms is important [16]. One security measure to consider is to make sure brokers and data sources are really who they claim to be. This can be accomplished by entity authentication. One of the ways to realize that is by attestation [26], which can be implemented using, for example, Intel Software Guard Extensions [27]. Sometimes, the brokers might not be trusted, and they might inject malicious code into the results of mashup execution, which would harm the end-user's machine. Guarding against untrusted brokers can be applied using encryption techniques on mashup results, such as the order-preserving encryption algorithm [28].

Reliability is an important aspect in networking systems, which is needed to make sure the system is still functioning under node failure or node leaving the network (churn). Node failure and churn both cause degradation of service. Addressing the reliability issue in our system can be realized by the following measures. First, the way to detect node churn or failure is by brokers sending heartbeat messages to their neighbors to make sure they are still alive. Second, if the broker that fails or leaves the system belongs to the Chord ring (i.e., group head), then the ring connections can be restructured, which is explained in [25]. However, if the leaving/failing broker is one of the brokers in a given unstructured group, then the broker can assign one of its direct neighbors as a deputy. This way, the broker periodically synchronizes with its deputy to make sure the deputy is hosting the same mashups hosted by the broker. Consequently, when a deputy broker discovers the absence of its neighbor, it takes over and notifies the group head of this change. Third, allowing several copies of a given mashup to be hosted at different brokers would alleviate the effect of node failure and churn, which is explained as follows. The genetic approach is already capable of hosting a given mashup on several brokers instead of one. The only thing needed is to change the constraint mentioned in Eq. (2) to reflect a higher number of copies of a given mashup in the network. This way, even when one of the brokers fails or leaves the network, there would be other brokers that host the same mashup, and they can satisfy the requests of end-users.

5 System Complexity

It is worthy to shed light on the complexity of our proposed system. Finding group heads in the system employs random walks, and the communication complexity for random walks is $O(N)$ where N is the number of brokers in the network, but this is assuming that the random walker has an infinite TTL value. However, in our work, a random walk is constrained by a TTL value. Consequently, the communication complexity in our case becomes $O(TTL)$. This effort is only needed at the initialization of the architecture. After the choice of group heads is settled, the group heads join the Chord ring, and this join operation has a communication complexity of $O(log^2\ R)$ where R is the number of brokers in the Chord ring [25], and this operation is needed every time a broker joins the Chord ring.

After that, each group head would build its unstructured group using sequence number flooding or reverse path flooding. In sequence number flooding and reverse path flooding, each node is visited at most two times, one time in the forward path and another time to report back to the group head. So, each edge is visited at most twice, which is considered linear complexity. The number of messages initiated by sequence number flooding is less than that for reverse path flooding, which is due to not allowing a message with a previously received sequence number to be propagated further. Accordingly, both approaches have a linear communication complexity $O(J)$ where J is the number of edges in the network. This is needed when the architecture is initialized and when a new broker joins the unstructured groups.

Deciding which broker hosts a given mashup is determined by the genetic algorithm, and its time complexity is affected by the number of generations, the size of the population, and the length of the chromosome. In our work, a new generation emerges after each iteration. Also, the length of the chromosome equals the number of brokers multiplied by the number of mashups. Therefore, the time complexity of the genetic approach is $O(T \times P \times N \times M)$ where T is the number of iterations, P is the population size, N is the number of brokers in the network, and M is the number of mashups. This cost is only needed when a new mashup is introduced to the system.

When a user requests the execution of a given mashup, then a lookup operation for the mashup is performed on the Chord ring and it takes $O(log\ R)$ communication complexity [25], then after reaching the group head, the request is forwarded to the broker responsible of hosting the mashup and this step requires at most H messages where H is the diameter of the unstructured group which the group head is responsible for; so this communication complexity is translated as $O(H)$. Then, the mashup is executed at that broker, and the result is returned via the reverse path to the end-user.

6 Results

We evaluate our proposed system via simulation on a Core i5 computer with a 2.4 GHz CPU speed and 8 GB RAM. Our experiments are performed on a network that we extracted from the Autonomous Systems "as20000102" network [29], which is part of the Stanford Network Analysis Project (SNAP) [30,31]. The Stanford Network Analysis Project (SNAP) provides several types of realistic networks, such as those used for social media. Since we are building an architecture that relies on peer-to-peer networks, we needed a network that represents peers on the Internet, which is exactly what the Autonomous Systems network represents. We employed the GraphStream dynamic graph library [32] to make sure that the extracted network does not contain any isolated subnetworks. The diameter of the extracted network is 8, and it contains 1050 nodes such that the number of end-users is 200, the number of data sources is 100, and the rest of the nodes are brokers. As much as possible, data sources and end-users are chosen among the nodes that have a degree of 1, which indicates that they reside on the edges of the network. To make sure our results are valid, we repeat each experiment 20 times and compute a 95% confidence interval, which we plot on our charts.

In the experiments, we use default values of a few parameters unless otherwise stated. The default value for the number of groups is 3, the crossover operator is single-point crossover with crossover probability 0.8, the mutation operator is bit flip mutation with mutation probability 0.8, the number of iterations is 1000, the population size is 100, the tournament size is 2, and the number of mashups is 150. Link delays and bandwidth values are distributed on network edges using a uniform distribution such that the average delay for network edges is 20 ms and the average bandwidth for network edges is 30 Kbps. The size of each data source is also drawn from a uniform distribution such that the average size of data sources is 70 Kb.

The way we build mashups is by randomly picking at most two data sources per mashup and then randomly selecting other operators to work on the data, such as filtering and sorting, such that the number of operators per mashup is 7. It is worth mentioning that the filter operators are simulated by assigning a filter percentage to each filter operator. We compare several combinations of approaches and architectures; these are listed as follows.

- Sequence-GA: Our hybrid architecture, in which groups are formed using sequence number flooding, and mashup hosting is decided by using genetic algorithms.
- Reverse-GA: Our hybrid architecture in which groups are formed using reverse path flooding, and mashup hosting is decided using genetic algorithms.
- Sequence-Greedy: Our hybrid architecture, in which groups are formed using sequence number flooding, and mashup hosting is decided using the greedy approach.

- Reverse-Greedy: Our hybrid architecture, in which groups are formed using reverse path flooding, and mashup hosting is decided using the greedy approach.
- Sequence-Random: Our hybrid architecture in which groups are formed using sequence number flooding, and mashup hosting is decided using the random approach.
- Reverse-Random: Our hybrid architecture in which groups are formed using reverse path flooding, and mashup hosting is decided using the random approach.
- Structured: A completely structured architecture which follows Chord. It is identical to the structured part that we explained in this work.
- Unstructured: A completely unstructured architecture where mashup hosting is decided by randomly selecting brokers to host mashups. Also, mashup search is performed using sequence number flooding.

Fig. 6 shows the distribution of the different operators we used in mashups. We can see that the filter operator has the highest proportion since data filtering is potentially the most common task for end-users.

The next experiment pertains to the part in which the genetic algorithm is employed to determine which brokers would host mashups in the unstructured network. So, in Fig. 7, we test the effect of changing the number of iterations on the ability of the genetic algorithm to map mashups to brokers that result in minimal delays. The number of iterations pertains to only the genetic approach, so the only approaches we compare here are Sequence-GA and Reverse-GA. We can see that when the number of iterations increases, the genetic algorithm has a greater chance of finding better mashup hosting solutions, which yields minimal average delays in executing mashups. Also, we can see that the Reverse-GA generates solutions with lower delays than Sequence-GA. The reason behind that is that in Reverse-GA, when a broker receives a message with an identical sequence number coming from a different path, then it checks the recorded accumulative delay in the received message and compares it with the delay it previously recorded for reaching the group head. As a result, if the delay associated with the new message showed less delay than the delay recorded at the broker, then the broker records the new delay and reports that to the group head via the reverse path. In contrast, with Sequence-GA, the broker realizes that this message was previously received from a different path before, so, it simply ignores it and does not allow it to propagate further in the ongoing flooding process.

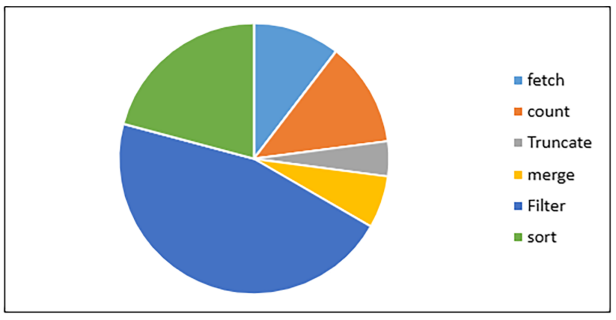

Figure 6: A chart showing the distribution of the different types of mashup operators used in the built mashups.

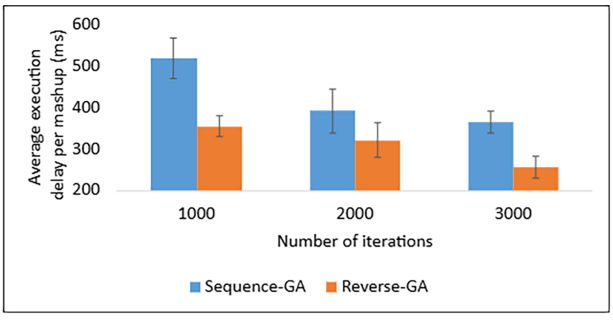

Figure 7: Average delay of executing mashups using the genetic algorithm while the number of iterations varies.

In the next experiment, we study the outcome of changing the crossover probability on the effectiveness of the genetic algorithm to find a mashup hosting solution with less mashup execution delays. Also, here, the crossover percentage affects solely the genetic approach, so the only approaches in conversation are Sequence-GA and Reverse-GA. Fig. 8 implies that using a higher crossover probability generates mashup hosting solutions with less average delay of executing mashups, which is logical because a higher crossover probability means that when two solutions are chosen for mating, then there would be a higher chance for

crossover to be performed and thereby generating children with different genes than their parents and hence generating different mashup hosting solutions. This adds to the variety of solutions in the solutions pool and accordingly helps the algorithm to move faster toward generating higher-quality mashup hosting solutions with lower delays. Again, the Reverse-GA allows for generating solutions with lower delay values due to the ability of reverse path flooding to find minimal paths between group heads and the brokers in their unstructured groups.

In the next two experiments, we do not include the completely structured and the completely unstructured approaches in the experiment, so that we focus the comparison between the different flavors of our hybrid approach. For the next experiment, we evaluate the average delay of mashup execution in the different approaches while the number of groups changes. So, we can notice in Fig. 9 that the increase in the number of groups results in generating solutions with lower delay. This is logical because when the number of groups increases, then the number of brokers existing in each group becomes smaller. This way, the search space on which Sequence-GA and Reverse-GA need to explore becomes smaller, and therefore, they would be able to find better solutions with fewer iterations. For Sequence-Greedy, Reverse-Greedy, Sequence-Random, and Reverse-Random, there is no clear relationship between the delay and the increase in the number of groups. Nevertheless, increasing the number of groups would allow those approaches to operate in different parts of the topology, which may sometimes result in lower delays. Further, we can see that all the approaches that employ reverse path flooding outperform their counterpart approaches that utilize sequence number flooding. This is reasonable since the flooding process in Sequence-GA can be cut earlier because of receiving messages with identical sequence numbers, whereas Reverse-GA allows for an extended flooding process that leads to finding paths with minimal delays between group heads and the brokers in their unstructured groups. Furthermore, we notice that Reverse-GA and Sequence-GA are superior to Reverse-Greedy and Sequence-Greedy approaches. This is because Reverse-GA and Sequence-GA rely on genetic algorithms in deciding where to host mashups, and this is a more effective way than depending on the greedy approach. The greedy approach guides the search toward the direction of the current best solution, and that may falsely pull the search away from other directions where the brokers with the lowest delays of hosting mashups exist. However, Sequence-Greedy and Reverse-Greedy approaches are still better than Sequence-Random and Reverse-Random approaches, which is due to the blind mashup to broker placement conducted by the random approach, resulting in higher delays.

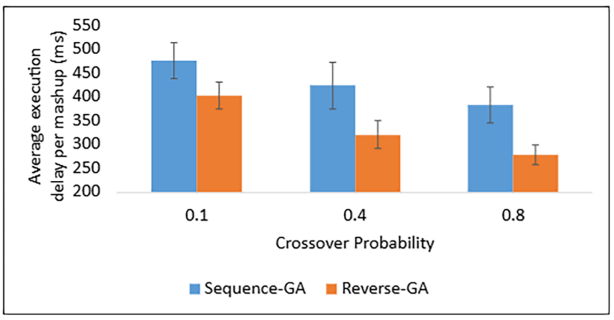

Figure 8: Average delay of executing mashups in the solutions found by the genetic algorithm while crossover probability varies.

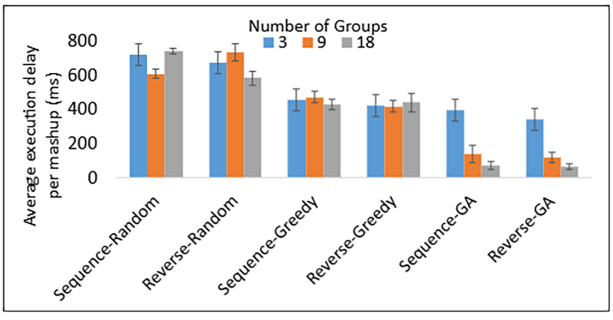

Figure 9: Average delay of executing mashups in the different approaches while we vary the number of groups in the experiment.

Next, we evaluate the computer execution time per group needed to execute the different approaches. We notice in Fig. 10 that having more groups indicates that the number of brokers within a single group becomes

small, and consequently, less time is needed to form a single group. In addition, all the approaches that employ reverse path flooding need less execution time to form groups. This is rational because the sequence number flooding works such that if a given broker receives a message with the same sequence number as a previously received message, then it won't allow the message to continue its way in the flooding process. Another observation is that Sequence-Random and Reverse-Random approaches execute fast, which is a direct consequence of the random placement of mashups on broker nodes, which takes less time to conclude. However, Sequence-Greedy and Reverse-Greedy require more time than the random approach due to the greedy approach advancing the search process to look for brokers with less delay to host mashups. In contrast, Sequence-GA and Reverse-GA both demand more execution time, and this is due to the nature of the evolutionary approach that depends on iterations for improvement. Having said that, the high execution time of Sequence-GA and Reverse-GA is tolerable because each of them is only needed during the period of deciding where to host mashups, while subsequent requests to execute those mashups enjoy the efficient paths found.

In Figs. 11 and 12, we aim to compare the different architectures, namely, completely structured, completely unstructured, and the two flavors of our hybrid architecture (Reverse-GA and Sequence-GA). Fig. 11 depicts the delay of executing mashups in the different architectures. On the one hand, the completely unstructured approach is the worst, which is intuitive because it randomly hosts mashups on brokers without intentions to minimize delay. On the other hand, the completely structured approach follows the Chord structure, which is known for fast lookup operations, which is better than the unstructured approach. However, the strict structure of Chord does not leave any room for optimizing delays. In contrast, Reverse-GA and Sequence-GA rely partially on the Chord architecture in the structured part, and then they make use of the genetic approach in the unstructured part, which looks for effective minimization of delays. Hence, they beat the completely structured approach. It is worth mentioning that there is no error bar for the structured approach because it follows a very strict structured topology, which always guides the mashup hosting decisions in the same direction, and that leads to the same behavior and the same results in each run.

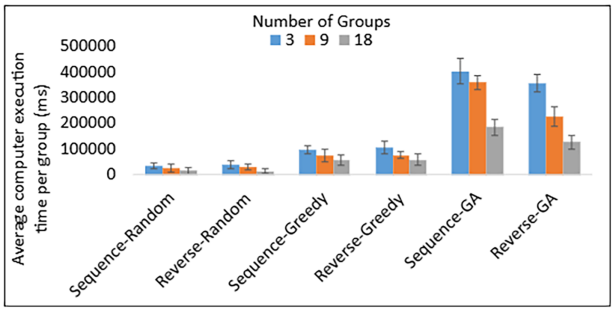

Figure 10: Average computer execution time needed to form a single group in the unstructured part when the number of groups changes.

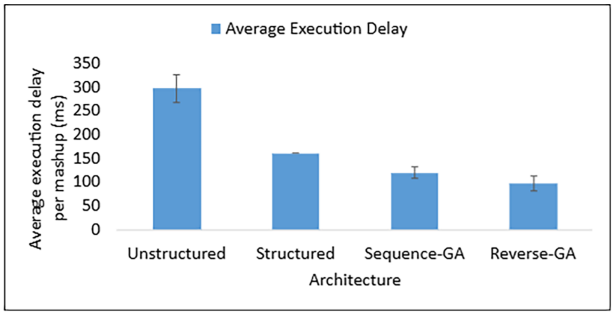

Figure 11: Average delay of executing mashups for the completely structured, completely unstructured, and the two flavors of our approach.

Fig. 12 shows the computer execution time for the different architectures. The completely structured outperforms the rest of the architectures due to the strict structured approach that behaves according to the Chord ring, and it is known to be fast at the expense of not generating the minimal delays (as previously shown in Fig. 11). The completely unstructured approach, Reverse-GA, and Sequence-GA generate comparable computer execution time. This is explainable because the high execution time of the unstructured approach is due to using the flooding approach for performing lookups for mashups, which

is a time-consuming process. Also, for Sequence-GA and Reverse-GA, they both use flooding in addition to using the genetic algorithm, which keeps improving delay values by repeated iterations, and this results in a high computer execution time. However, we mentioned before that the high execution time needed by Reverse-GA and Sequence-GA is only needed during the time of determining the brokers that will host mashups, and it won't affect executing mashups after that.

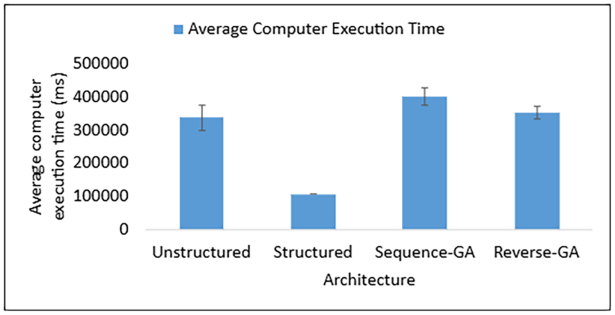

Figure 12: Average computer execution time for the completely structured, completely unstructured, and the two flavors of our approach.

Current workflow automation and orchestration platforms appearing in industry, such as PipesDigital [4] and IFTTT [5], or the ones appearing in research, such as the work in [6], are based on centralized architecture. Therefore, adopting a distributed architecture like the one we propose would enhance their scalability. In addition, we showed in this section that our routing approach, which is achieved in a hybrid architecture, generates lower mashup execution delays than pure structured and pure unstructured approaches. Therefore, the current platforms can employ this hybrid approach to minimize workflow execution delay.

7 Conclusion

Decentralized architectures support scalable information-centric platforms. In this paper, we proposed a hybrid approach to design a decentralized mashup platform relying on both structured and unstructured peer-to-peer networks, which is an answer to the research question **RQ1**. The structured part utilized Chord to build a virtual ring topology. We employed sequence number flooding and reverse path flooding to build groups of brokers that are connected via an unstructured peer-to-peer overlay. Consequently, relying on both structured and unstructured topologies on one hand and employing reverse path flooding in forming groups on the other hand allowed for efficient message routing, which is an answer for the research question **RQ2**. Further, we made use of genetic algorithms to decide which brokers would host and execute mashups with minimal execution delays. Our results are promising, and they show that the flavor of our scheme that uses reverse path flooding generates less mashup execution delays than other flavors that utilize sequence number flooding, random mashup placement, and greedy mashup placement. Our approach also outperformed the pure structured and unstructured approaches. We discussed how our system can be protected from node failure and churn. We also pointed out security measures that can be followed to enhance our system. However, these issues are not discussed in detail, and therefore, they are considered a limitation of this work, and we aim to investigate them as part of future work. Another part of the future work is to evaluate other optimization techniques, such as particle swarm optimization and ant colony optimization, in finding mashup hosting solutions in the unstructured part of our proposed architecture.

Acknowledgement: Not applicable.

Funding Statement: This study has no funding support.

Author Contributions: The authors confirm contribution to the paper as follows: conceptualization, Osama Al-Haj Hassan; methodology, Osama Al-Haj Hassan and Ammar Odeh; validation, Ammar Odeh and Abdullah Aref; investigation, Osama Al-Haj Hassan and Ammar Odeh; data curation, Abdullah Aref and Ghassan Samara; writing—original draft preparation, Osama Al-Haj Hassan, Ammar Odeh, Abdullah Aref and Ghassan Samara; writing—review and editing, Osama Al-Haj Hassan and Ghassan Samara; visualization, Abdullah Aref and Ghassan Samara; supervision, Osama Al-Haj Hassan. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: Data available on request from the Corresponding Author, [Osama Al-Haj Hassan], upon request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Huang A, Huang N, Hong Y. Workflow automation in open-source software development: accelerating innovation through mechanization and orchestration. Inf Syst Res. 2026.
2. Wu Q, Zhou Q, Sun R, Zhou M, Feng J, Guo T, et al. Blending serverful and serverless cloud resources for cost-effective workflow execution. IEEE Trans Autom Sci Eng. 2026;23:9925–36.
3. Alam KA, Haroon M, Ain Q, Inayat I. A data-driven API recommendation approach for service mashup composition. Int J Syst Assur Eng Manag. 2025:1–22.
4. Pipes Digital. [cited 2026 May 18]. Available from: https://www.pipes.digital/.
5. IFTTT. [cited 2026 May 18]. Available from: https://ifttt.com/.
6. Trinh TD, Wetz P, Do BL, Kiesling E, Tjoa AM. Distributed mashups: a collaborative approach to data integration. Int J Web Inform Syst. 2015;11(3):370–96. doi:10.1108/IJWIS-04-2015-0018.
7. Xu Y, Zhang S, Gao H, Yin Y, Hu J, Li R. Explainable service recommendation for interactive mashup development counteracting biases. Inf Sci. 2025;708:122049. doi:10.1016/j.ins.2025.122049.
8. Sun M, Xu Y, Tan Z, You D, Chen Z. Multi-level graph contrastive learning for cold-start recommendation in mashup development. Inf Sci. 2025;717:122319. doi:10.1016/j.ins.2025.122319.
9. Deng S, Wu H, Taheri J, Zomaya AY, Wu Z. Cost performance driven service mashup: a developer perspective. IEEE Trans Parall Distrib Syst. 2015;27(8):2234–47. doi:10.1109/TPDS.2015.2482980.
10. Chen Z, Kim M, Cui Y. SaaS application mashup based on High Speed Message Processing. KSII Trans Internet Inform Syst. 2022;16(5):1446–65. doi:10.3837/tiis.2022.05.003.
11. Sehar U, Ghazal I, Mansoor H, Saba S. A comprehensive literature review on approaches, techniques & challenges of mashup development. Int J Scient Eng Res. 2022;13(3):383–397.
12. Shiddiqi AM, Prasamya S. Shortest path determination using google mashups technology. In: International Conference on Informatics for Development (ICID 2011); 2011 Nov 26; Yogyakarta, Indonesia.
13. Yan C, Zhong W, Zhai D, Khan AA, Gong W, Xu Y, et al. Popularity bias in correlation graph-based API recommendation for mashup creation. ACM Trans Intell Syst Technol. 2024;16(1):1–18.
14. Cao B, Peng M, Xie Z, Liu J, Ye H, Li B, et al. PRKG: pre-training representation and knowledge-graph-enhanced Web service recommendation for Mashup creation. IEEE Trans Netw Serv Manag. 2024;21(2):1737–49.
15. Alhosaini H, Alharbi S, Wang X, Xu G. API recommendation for mashup creation: a comprehensive survey. Comput J. 2024;67(5):1920–40. doi:10.1093/comjnl/bxad112.
16. Yoshihama S. Security for mashups. In: Encyclopedia of cryptography, security and privacy. Cham, Switzerland: Springer; 2025. p. 2291–3.
17. Bamhdi A. Evaluation of end-user web mashup development. Comput Inform. 2024;4(2):112–29.

18. Verma A, Thakur S, Kumar A, Prasad Mahato D. Routing algorithm for sparse unstructured P2P networks using honey bee behavior. Int J Commun Syst. 2025;38(2):e5978. doi:10.1002/dac.5978.
19. Cruciani A. Maintaining a bounded degree expander in dynamic peer-to-peer networks. arXiv:250617757. 2025.
20. Legheraba MA, Potop-Butucaru M, Tixeuil S, Fdida S. Emergent peer-to-peer multi-hub topology. In: 2024 22nd International Symposium on Network Computing and Applications (NCA). Piscataway, NJ, USA: IEEE; 2024. p. 219–26.
21. Lim A, Jang S, Lee J. P2P-Fed: a decentralized federated learning platform on structured peer-to-peer systems. In: 2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid). Piscataway, NJ, USA: IEEE; 2025. p. 124–33.
22. Roy I, Mitra R, Rahimi N, Gupta B. Efficient non-DHT-based RC-based architecture for fog computing in healthcare 4.0. IoT. 2023;4(2):131–49.
23. Ching CW, Chen X, Kim T, Ji B, Wang Q, Da Silva D, et al. Totoro: a scalable federated learning engine for the edge. In: Proceedings of the Nineteenth European Conference on Computer Systems. New York, NY, USA: ACM; 2024. p. 182–99.
24. Samimi A, Goudarzi M. A novel strategy for optimal data replication in peer-to-peer networks based on a multi-objective optimisation–NSGA-II algorithm. IET Netw. 2025;14(1):e12141. doi:10.1049/ntw2.12141.
25. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H. Chord: a scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Comput Commun Rev. 2001;31(4):149–60.
26. Ménétrey J, Grüter A, Yuhala P, Oeftiger J, Felber P, Pasin M, et al. A holistic approach for trustworthy distributed systems with WebAssembly and TEEs. arXiv:231200702. 2023.
27. Wang Z, Zhou Y. Analysis and evaluation of intel software guard extension-based trusted execution environment usage in edge intelligence and internet of things scenarios. Future Internet. 2025;17(1):32.
28. Pei J, Shi Y, Feng Q, Shi R, Lan L, Yu S, et al. An efficient confidentiality protection solution for pub/sub system. Cybersecurity. 2023;6(1):34.
29. Leskovec J. Autonomous systems dataset. [cited 2026 May 18]. Available from: https://snap.stanford.edu/data/as-733.html.
30. Leskovec J, Sosič R. SNAP: a general-purpose network analysis and graph-mining library. ACM Trans Intell Syst Technol (TIST). 2016;8(1):1–20.
31. Leskovec J, Krevl A. SNAP datasets: stanford large network dataset collection. 2014 [cited 2026 May 18]. Available from: http://snap.stanford.edu/data.
32. University of Le Havre. GraphStream: A dynamic graph library. Available from: https://graphstream-project.org/. [Accessed: 2026 May 18].