Viability of Dew Computing for Multilayered Networks

David Fisher, Stefan Gloutnikov, Yaoyan Xi, Sadib Khan

Department of Computer Science
San José State University
One Washington Square
San Jose, California 95192

Abstract

In the ever-changing world of technology, it can be difficult to distinguish one idea from the next. New paradigms are an excellent example, and this paper explores one of the more recent developments in the networking industry—dew computing. Sometimes called local computing, microclouds, and even four-tier architecture - these ideas are all under the same umbrella as dew computing. We invite our readers to take the journey with us, as we delve into what makes dew computing a novel innovation in the modern tech industry. After a brief background on the topic, we propose a new design of cloud-dew architecture that seeks to improve upon gaps found in previous iterations. Our design intends to bring the strongest elements of cloud, fog, dew, and peer-to-peer computing into one cohesive, high-availability system.

Keywords:

Dew computing; multilayer architecture; data consistency; latency; Internet computing paradigm; availability; scalability

1. Introduction

Cloud computing is a well-established paradigm in modern computing that brings many benefits, like universal access, scalability, and many others to the user. While we are able to take advantage of all these benefits, it also introduces a new challenge—all of the resources that a user uses are far away and not under the user's control. If the network connection is lost, the user loses all access to the service and the data. There is also a significant amount of latency introduced when the user accesses the cloud, that can hinder the user's experience to a certain degree. It is possible that the data the user is trying to access is physically located far away, and even with modern networking technologies the user can still experience slow access times. Dew computing is a new architecture and a new area of research proposed as a possible solution to the aforementioned problems [1].

The goal of dew computing is not to replace local (the user's machine) and cloud computing, but rather work as a middle-man to realize the full potential of both architectures. One scenario that quickly emerges is that by being present on the user's local machine, data, and services can still be accessed even without an internet connection, thus reducing dependency on the network. When an internet connection is again restored, the dew computing architecture will begin synchronizing data with the services provided by the cloud. Something else that we are seeing more often after the boom in cloud computing is that even though local machines are becoming more powerful, we are using them more often as merely an access port to the internet and cloud services. It is becoming more common to see the larger storage and processing power end up being underutilized or wasted by sitting idle. Dew computing can help bridge this increasing gap, while also keep pushing innovation to bring more powerful and cheaper computers to the consumers.

Later in this paper we look at previous related works in the field, as well as discuss in more detail the definition of dew computing. The paper follows up with several challenges that arise in dew computing and proposes areas improvements to address these challenges. We look at improvements on the remote server-side as well as the local client-side architectures in order to utilize excess user CPU cycles and replicate data that is usually stored in the cloud.

2. Related Work

Dew computing is a relatively new area of research, yet already we are seeing lots of publications driving the subject area forward. Along with strengthening the definition of dew computing, a breakdown of categories within dew computing and real-life applications are proposed in [16]. The authors propose looking at the dew computing paradigm within the following categories: Web in Dew (WiD), Storage in Dew (STiD), Database in Dew (DiD), Software in Dew (SiD), Platform in Dew (PiD), Infrastructure as Dew (IaD), and Data in Dew (DaD). We can see examples of already existing applications that fall within some of these categories. For example, Dropbox can be seen as following the dew computing paradigm and the STiD category--storage in dew has a cloud copy. GitHub within the PiD category--SDK and projects have a cloud copy. Mobile app stores like Google Play and the Apple App Store within the SiD category--software settings and ownership have a cloud copy [2].

On the Internet of Things (IoT) front, the authors in [3] propose a novel approach of bringing the dew computing paradigm to IoT devices, enabling more efficient streaming, storing, and collection of data. The implementation and study of a horizontally scalable balancer for dew computing services is presented in [4].

2.1. Dew Computing

One of the most fascinating applications of client-server architecture, the very bedrock of nowadays internet technology, is cloud computing. First introduced by Sasikala [5] and Rimal et al. [6], the cloud computing architecture became prevalent quickly thanks to its on-demand nature and low maintenance cost. Cloud computing makes resources and data more accessible, fosters more collaboration, and allows a more flexible and scalable development process. However, cloud computing requires a constant connection with the internet and communication, which is neither guaranteed nor efficient considering the overhead involved besides the security and privacy concerns. This is where the dew computing comes into the picture.

In 2015, the concept of dew computing was introduced by Wang [7] as a solution to address some of performance and logistical issues of cloud computing. Figure 1 demonstrates a common view of dew computing architecture.

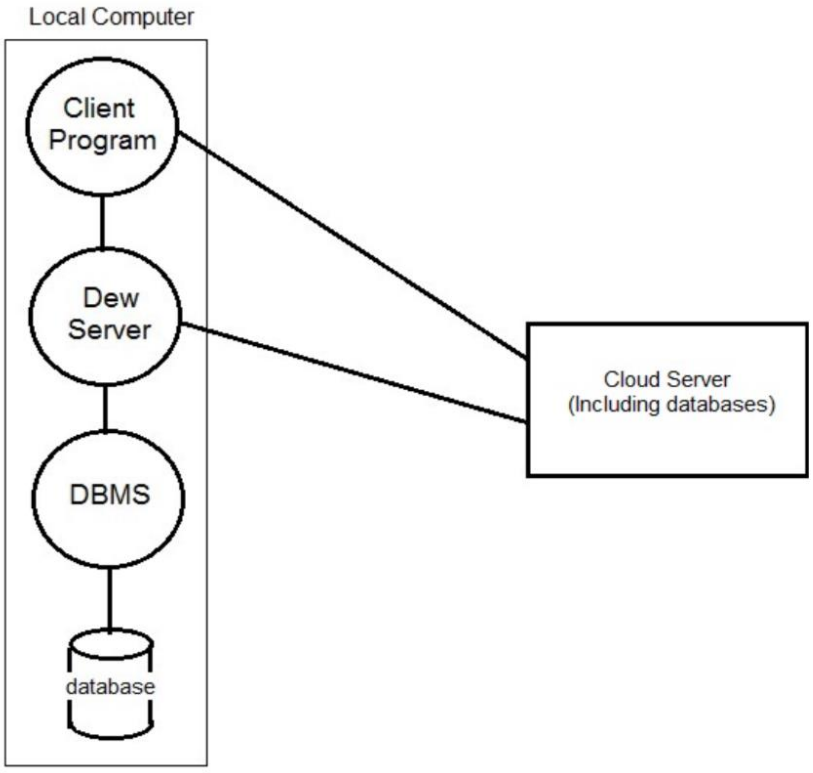

Figure 1. Dew computing components

Contrasted to conventional server and database in the cloud computing, a dew server (or client-side server) can reside inside a local computer, a smart phone, or a tablet to carry out the web browsing and other services locally when offline or to synchronize with the cloud server when online. This will not only alleviate the workload of cloud server but also eliminate the

communication overhead by providing the services locally. To make the dew computing more versatile and powerful, a local database management system (DBMS) is also created to map the website scripts and corresponding website database. With dew computing, clients can enjoy the seamless web services and localized processing capacities even without connecting to the internet.

Thus far, the research on dew computing has focused on two areas: features and applications. The feature research investigates the relationship between cloud computing, fog computing, and dew computing. Application research is more obvious and has focused on Internet of Things (IoT) streaming, medical care, indoor navigation, and cyber-physical systems [8]. The cloud-dew architecture would be a good paradigm for the intelligent homes of the future [9]. With the increasingly empowered processors in smart devices nowadays, more and more dew computing applications will be possible and change our lives forever. This significance explains our choice of dew computing as the research theme for this paper.

2.2. Distributed Computing

Despite of all the novelty and advantages of dew computing, data consistency and latency are still the issues to deal with when client-side server and cloud server synchronize with each other once internet connection is established.

The issues of data consistency and synchronization latency are not dew computing specific. Instead, they stem from the client-server architecture that dew computing is built on. Data consistency, specially cache consistency, makes sure the local copy of data in the cache is consistent with the data stored in the cloud server and that all the nodes see the same data. To this end, writing locking (concurrency control) mechanism or cache data validation process is in place. For dew computing, data consistency is theoretically easier to manage given the fact that only one client is served by a certain dew server compared with multiple clients attempt to read and write on the server simultaneously. Nevertheless, data and cache consistency is a great concern to consider.

Synchronization delays are another common issue with distributed systems, and they make the determination of order of events more challenging. The solution is a family of algorithms to determine the logical order of events to mitigate the synchronization problem.

3. Areas of Improvement

Existing cloud-dew architecture designs are strong in theory, but they have some pitfalls. For one, data consistency across multiple client nodes and remote servers has never been fully addressed. Very similarly to the challenges faced in cache consistency, we expect a solution to

this issue to be complex, yet entirely feasible. While fully-replicating cloud behavior locally sounds wonderful in theory, it has insurmountable roadblocks in practice. For example, a client machine is very unlikely to have the storage space and processing capacity required to simulate cloud behavior, even if it is serving only one user [7]. Not only are there hardware constraints, but there are also security, privacy, and ethical challenges to be considered. Full-replication of a cloud environment on a client machine is simply not feasible; however, partial replication might not be outside the realm of possibility. Existing cloud-dew solutions have yet to consider partial replication, both from a data perspective as well as a functional perspective.

Redundancy is also an important issue. The redundancy rate is an index that measures how much information on a node has a copy on the internet. It is defined as R=W/V, where R is the redundancy rate, V is the node's available information amount, and W is the amount of information inside V that has a copy on the Internet. While the redundancy rates for the mainframe and cloud computing stages are very low, it can significantly increase for the dew computing stage. If all the dew components have redundant copies on the Internet, the redundancy rate for dew computing would be 100% [8].

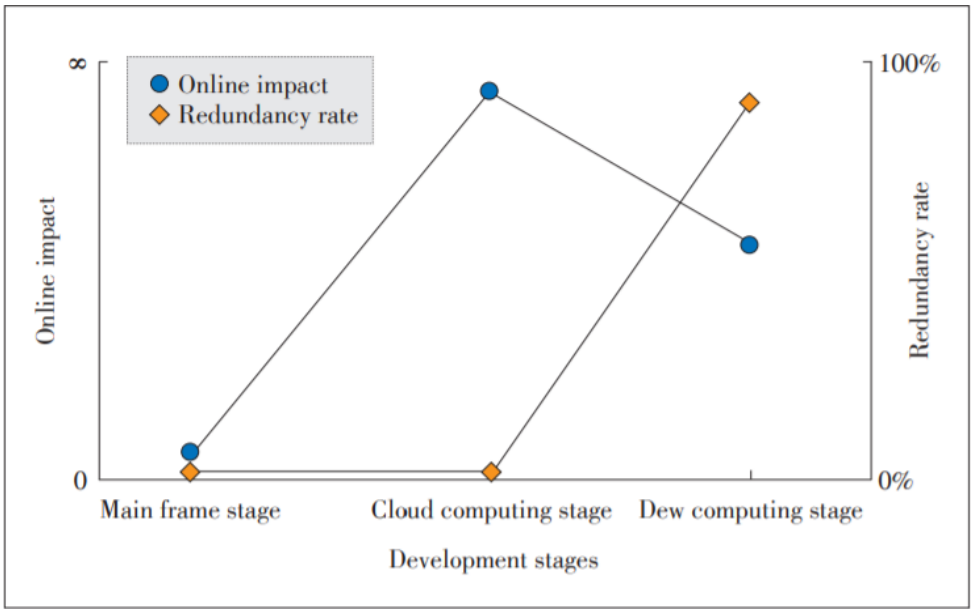

Figure 2. Online impact and redundancy rate change over major stages of Internet computing paradigm development [8]

4. Proposed Solution

In this section, we focus on the solution we developed during the project's solutioning phase. Our goal is to propose a viable architecture which addresses the issues mentioned in Section 3. Because this section is rather large, we have internally divided it into subsections to organize our

thoughts into the most readable format possible. Each subsection focuses on the solution from a different perspective.

We acknowledge that this section will be critical to the success of our project, and we expect the reader to consider gaps in our own solution. Therefore, we periodically challenge ourselves with hypothetical scenarios which we believe the reader may be pondering as well. Following these preemptive questions, we have listed our counterarguments, with the ultimate goal of further strengthening our proposed solution. Naturally, the focus of these scenarios will address the challenges listed in Section 3.

Due to the sheer size and complexity of such a new architecture, we have limited some of our discussion areas to the high and medium levels, simply due to project timeline constraints. However, we hope that our arguments are persuasive enough to convey that they are sound in theory and that a practical implementation is feasible.

At a high level, our solution revolves around two main ideas: (1) how to leverage excess client CPU cycles to perform meaningful work, and (2) how to replicate data, traditionally stored only in remote data centers (i.e., the cloud), across multiple client nodes, which are fully owned and operated by the end-user. The goal of this architecture design is to drastically increase web browsing speed and availability, without requiring additional investment in hardware. In other words, our proposed architecture is entirely software-defined. Figure 3 provides an architectural overview below.

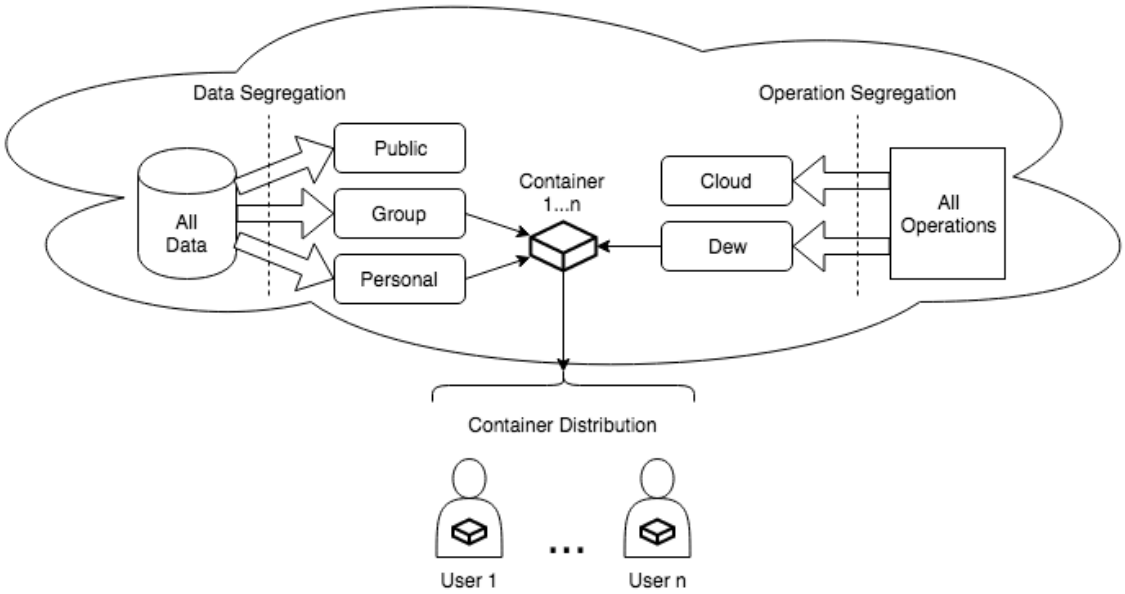

Figure 3. Architecture overview

4.1. Remote Server-side (Cloud) Architecture

Since cloud computing is a mature technology, we place an emphasis on what would need to be changed in order to support the new solution, as opposed to a detailed description of current cloud architecture. Cloud architecture remains relatively unchanged when compared to traditional setups, but includes four new, key aspects: (1) logical differentiation between consumers of data - as defined in the data segregation layer, (2) logical differentiation between functions which should be executed on the server versus functions which should be executed on the client machine - as defined in the operation segregation layer, (3) customized container creation and deployment on client machines, and (4) synchronization mechanisms to assimilate client changes into the server copy. Data and operation segregation, as well as container creation and deployment, are covered in separate subsections below, while details of the synchronization mechanism are covered in Section 4.3.

4.1.1. Data Segregation Layer

The data segregation layer serves to logically separate data into three categories, based on which type of end user has access to it. The first and most coarse-grained category is public, the second and most fine-grained category is personal, and the third category, group, is defined to be a middle-ground of the other two. Each datum hosted on the remote server must be placed into one of these three categories, which governs how it is to be deployed on client machines.

In order to categorize datum as either public, personal, or group, we start by choosing a number *G*. *G* represents the maximum number of clients that can belong to a group before the group is considered public. We also define *C* as the number of clients which have write access to a given datum. The categorization algorithm is as follows:

```
for each datum in a given data set
	if datum.C <= 1
		category = personal
	else if datum.C <= G
		category = group
	else
		category = public
```

The purpose of categorizing data is to control the level of effort required to maintain cache consistency. Personal data is very easy to keep consistent, as there is only one user who can modify the data, typically the creator but not always the case. On the other hand, public data is near impossible to keep consistent across multiple clients, as any of the *N* number of users can modify it with any *F* frequency. Trying to maintain cache consistency on public data would lead to an insurmountable overhead while attempting to maintain consistency across all *N* client machines. Group data is difficult to keep consistent, but it is possible with distributed token-based, mutual exclusion algorithms - provided we limit the maximum number of members and

promote a leader to announce updates to changed data [10]. This limitation is the sole purpose of defining *G* in the above categorization algorithm.

Note that data is categorized based only on users which have write access. Users with read-only access are not to be considered, because the proposed architecture operates on a hybrid model of weak and strong consistencies, where either one may be used depending on the user's write privileges. For a given datum and a given user, it is guaranteed that the level of consistency will be strong if the user has the ability to modify the datum. However, if the user can only read the datum, then the datum is to be considered weakly consistent. The goal of this hybrid model is to combine the extreme availability that a weakly consistent system offers with the reliability of a strongly consistent system when it matters most. This concept is discussed in much more detail in Section 4.2.2.

4.1.2. Operation Segregation Layer

Not all cloud operations are eligible to be hosted on a client's local machine. Unlike data center machines, client machines have memory and processing constraints; therefore, any operation which requires huge amounts of computational resources would be ineligible to host locally. Security and privacy constraints may be another localization limitation, as a user may not be authorized to access or modify data from another user. These are just a few examples of many; however, even with limitations, the ability to locally host a subset of cloud operations is an immensely powerful feature for both improving web response times for end users as well as reducing loads on cloud servers.

From a user's perspective, we expect up to ten times (10x) better website performance with as little as ten percent (10%) of cloud functionality migrated to local processing if the functionality is chosen properly. That is an efficiency rate of up to one hundred times (100x). Therefore, we introduce the operation segregation layer as the entity responsible for choosing operations most suitable for deployment on client machines.

The concept can be simple. For example, the list of all cloud-hosted functions is scanned for dependencies - the functions with either no dependencies or dependencies which are able to be supported by the client are segregated into a subset, which is deployed on the client's machine.

The concept can also be more complex. New functionalities can be created specifically for client-side deployment, or existing functionalities can also be modified. For example, Facebook's search function which has a dependency to access all user data would not be immediately eligible, but a modified search function might - one which only searched for the user's friends and friends-of-friends.

4.1.3. Container Creation & Deployment

The proposed architecture revolves around the concept of containerization, in that each client hosts customized containers created by domain owners (e.g., YouTube, Facebook, CNN, etc.). Since client-side machines are owned and operated by end users, there is no guarantee of homogeneity among hardware. Not only could one end user have a radically different operating environment than another end user, one end user could also have diversity among his/her own devices. For example, one user could have an iPhone and Chromebook; while a second user might own an Android and MacBook Pro.

Client operating systems are one restriction, but so is the availability of computing resources. It would be relatively easy to support a dew computing environment aimed at powerful workstations with 16GB of RAM, multi-terabyte hard drives, and unlimited power by consistently being plugged into an outlet. On the other hand, supporting a dew computing architecture on mobile devices is a much more challenging problem to solve. Mobile devices introduce a new requirement that our solution needs to account for - efficiency, both in terms of power consumption as well as storage.

Our solution suggests containerization as the answer to these challenges. Containers are the perfect vehicle for logically grouping website data and functionality into a single, deployable entity. Not only are they designed to be lightweight and mobile, containers also leverage existing features that are provided by the underlying operating system without requiring a dedicated operating system installation [11]. As previously mentioned, this is important because client machines have finite storage and computational resources; overhead must be kept to an absolute minimum.

Containers also allow dew computing development to be standardized to a single platform. Taking note of how Java solves the problem of mobility via the use of the Java Virtual Machine (JVM) [12], we believe dew computing should head toward a similar direction. After all, it would be unrealistic to expect domain (e.g., YouTube, Facebook, CNN, etc.) developers to write different code for every major operating system in the world. Container standardization would allow developers to write code once and deploy the code on any end-user device, regardless of the user's underlying architecture.

Once a remote cloud server has deployed a container on a client's local machine, the client has the ability to leverage the features provided by the container without requiring data transmittance over a network. The deployment of the custom container is a one-time activity; however, it must be mentioned that the data on the container must be kept in-sync with the data maintained on the server to avoid cache inconsistency. In order to avoid noticeable service disruption by the user, data synchronization is defined to be an asynchronous operation and is discussed in more detail in Section 4.3.

4.2. Client-side (Local) Architecture

Although our solution is entirely software-defined, there are still significant changes to client-side architecture. Our solution introduces new complexities in terms of accessing functionality and data, but drastically increases the speed, availability, and efficiency of resource utilization. We expect these benefits to be worth the tradeoff in complexity.

The biggest change is that data and functionality may be accessed locally, instead of remotely, via the container deployment discussed in Section 4.1.3. Remote integration is not removed, however; remote operations continue to play a key role. Collaboration between the local machine, the remote machine, and even other users' machines is the foundation we have built our solution upon. Section 4.2.1 stresses the collaborative aspect of our solution, while Section 4.2.2 addresses the challenge of maintaining distributed cache consistency.

4.2.1. Leveraging Localized Data & Functionality

The high-level goal of dew computing in general is to provide localized data and services without always requiring a connection to some remote server deployed in a data center. The local machine and the remote machine are to collaborate with one another to provide the end user with the best user experience possible. Localization substantially increases the availability and speed of operations because it eliminates the dependency on the network, but dew computing is not about completely eliminating the network - it is about leveraging it in a novel way.

As mentioned in Section 3, one issue with current dew computing solutions is that they are all-or-nothing, with no middle ground. It is either all the functionality and data replicated locally, or none of it. As discussed in Section 4.1.2, a better solution is to allow a hybrid approach, where some functions are performed locally while others are performed remotely. We expect the integration between local processing and remote processing to be both seamless and significantly faster than remote-only processing.

When users access a website, they should have no idea where the processing and data consumption is being performed. They should only know that it works as expected with a high degree of responsiveness. When the container is deployed on a client's machine, there should be some level of location transparency [10], so that development of such a hybrid system is not overbearing.

For example, a function *loadHomePage()* should be written in such a way that the underlying complexity is hidden from the developer and is applicable for dew computing users as well as traditional cloud-based users. The function *loadHomePage()* might return a local file for users with the page and functionality present on their local machine, or it might return a copy of the

remote file for users without a dew computing architecture installed on their system. RPC is the perfect mechanism for supporting a hybrid web environment [10].

We envision a future where a user goes to their browser, types in a URL, hits enter, and the page loads within milliseconds. The user can click around and interact with the site the same way that they always have, but it just seems so much *faster*. Some operations are being executed on the user's local machine, while other are being executed way off in a remote data center, all at the same time. Some data may be modified locally, while some other data might be modified directly on the cloud. The end user has no idea what is going on under the hood, but they are enjoying every moment of the browsing experience.

4.2.2. Data Modification & Cache Consistency

Maintaining cache consistency across multiple nodes in a distributed system has been a challenge researchers have been trying to solve for years [13][14][15][16]. Latency and unreliability in the network is the main culprit, but availability and clock synchronization among nodes in a cluster certainly also play a key role in the lack of development of a viable, cost-effective solution [13][16]. To date, a small handful of solutions have been proposed; however, they all have assumptions which are either very expensive to ensure (both monetarily and computationally), or require a very controlled environment [15]. In an environment which relies on the Internet for data transportation - such as dew computing - these solutions fall flat. A viable solution for ensuring data consistency in a multi-user, heterogeneous, Internet-scale environment needs to be both cheap and tolerant to uncontrollable diversity.

In essence, dew computing presents an almost identical challenge to that of distributed cache consistency. Many users on many machines may have access to the same data, and everyone needs to be able to trust the data's stability, consistency, and integrity [10]. After all, one of the primary goals of dew computing is to provide true data replication to end user devices. So how do we ensure cache consistency in a scalable manner? We know that we need to avoid introducing scenarios where the cost of maintaining cache consistency is too great (scalability), but we also know that caching still needs to be a focus in any dew computing solution (availability). Our solution addresses both of these aspects by attacking the challenge from a new angle - data categorization.

In Section 4.1.1, we introduced the concept of categorization of data within the data segregation layer. On the remote server, each piece of data is categorized into one of three groups - public, group, or personal - depending on how many users may modify it. We also mentioned that our data model is derived from both strongly consistent *and* weakly consistent models. In this section, we discuss details of how the concept works, as well as why we chose to leverage strengths of both models.

Here we introduce one more dimension of categorization based on the privileges of a given user. For each piece of data, the user has access to, there are two possible buckets it may be placed into: (1) the strong consistency bucket, or (2) the weak consistency bucket. If the user may modify the data (i.e., has write privilege), then that data is placed into the strong consistency bucket. If the user cannot modify the data (i.e., has read-only privilege), then that data is placed into the weak consistency bucket.

Once the entire set of data has been segregated into either the strong or weak consistency buckets, the category of the data is inspected. Based a given datum's category and bucket, a different caching policy will be employed, as depicted in Table 1.

	User has write privilege [Requires Strong Consistency]	**User has read-only privilege [Allows Weak Consistency]**
Public data	Read: Remote only Write: Remote only	Read: Local ~~Write: N/A~~
Group data	Read: Local Write: Group leader	Read: Local ~~Write: N/A~~
Personal data	Read: Local Write: Local	Read: Local ~~Write: N/A~~

Table 1. Caching policies

By employing different caching policies, we can guarantee scalability and consistency while providing the *fastest reads possible*. Local operations are always preferred over remote operations, except when it may introduce inconsistency into the environment. It has been observed that read operations are far more prevalent in the World Wide Web than write operations, so we expect that the majority Internet traffic can, in fact, be localized in a dew computing system [17][18]. However, for the write operations that do occur, our proposed solution continues to be applicable.

Each policy has been carefully chosen, with the reasonings as follows.

Public data which the user may modify:

- Many other users may modify the data concurrently, so an entity is required to coordinate global ordering, provide lock mechanisms, and act as a centralized source-of-truth. With an infinite number of potential users, the only feasible choice for this entity is the domain's remote server (i.e., the cloud).
- It is impossible to guarantee that the user has the latest changes cached locally. Frequent updates are likely to occur on public data. By the time the changes are sent over the network, the data may have already changed.

- Strong data consistency is required, as the local machine can factor the current data value into the modification event (e.g., increment an integer by one).
- A good example of public data which the user may modify is game state in a massively multiplayer online (MMO) game, such as World of Warcraft.

Public data which the user cannot modify:

- Many other users may frequently modify the data at any time, but the local machine does not require the latest state, only a *consistent* one.
- It is okay to not have the latest data, but the data should eventually be updated. This will happen during the periodic, asynchronous synchronization mentioned in Section 4.3.
- Weak consistency is acceptable, and it allows the user to leverage localization for data access without network connectivity.
- Good examples of public data which the user cannot modify are news feeds such as CNN.com and CNBC.com.

Group data which the user can modify:

- Like public data, group data can be modified by two or more members concurrently, and therefore requires an entity to coordinate global ordering, provide lock mechanisms, and act as a centralized source-of-truth. Unlike public data, there are a finite number of members in a group and therefore the cost of maintaining data consistency is controlled and predictable.
- In Section 4.1.2, we classified group data as data which can be modified by a finite, controlled number of users. If the user has write access to group data, then that means the user is a member of that group. One member of the group is elected to be the leader via classical election algorithms such as the Bully or Invitation algorithms [10]. The leader of the group is responsible for coordinating global ordering, providing lock mechanisms, and acting as a source-of-truth among all other group members.
- With this proposed group concept, we have narrowed the cache consistency challenge to a much more manageable number of users.
- When a group member wants to modify group data, they first request a lock from the group leader. Once the lock is retrieved, the member immediately returns the lock to the leader, including details of the change being made. Before the lock is made available again, the leader announces the change to all group members via reliable multicast [10]. Once the reliable multicast is received by all members, all members commit the change locally and the group leader makes the lock available again. Only the group leader is required to synchronize group data changes to the remote cloud server. This sequence of events is visualized in Figure 4.
- Group leaders may be re-elected. Group leaders are not to be a single point-of-failure.
- We acknowledge that this group communication scheme may actually be more expensive than making the change on the remote server, but we refute this as an issue in our

solution because: (1) local reads are not affected, (2) group synchronization is asynchronous to read operations, meaning that reads will still be incredibly fast, and (3) read operations are expected to be more frequent than write operations.

- Examples of group data which the user can modify are collaboration software such as Google Docs.

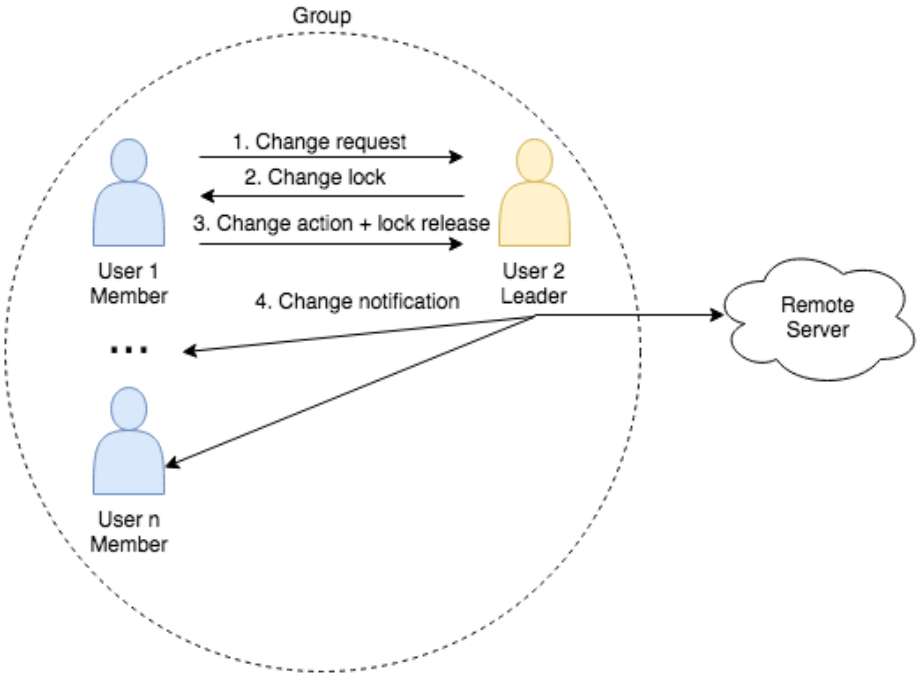

Figure 4. Group data modification

Group data which the user cannot modify:

- Group data which the user cannot modify has the same policy and reasoning as public data which the user cannot modify. Slightly outdated data is acceptable and will be periodically updated during synchronization.
- The user is not considered as a group member and therefore does not communicate with the group leader or any other member of the group.
- An example of group data which the user cannot modify is an observer in a team-based online game.

Personal data which the user can modify:

- We expect a significant portion of write operations to be on personal data, which is great because consistency is very easy to maintain. Only one user can modify personal data, and because of this trait, the user can freely modify their local copy without any additional overhead. This kind of data is by far the fastest for write operations.
- The user is the only entity responsible for synchronizing modifications to the remote cloud server.

- Examples of personal data are plentiful, and include configuration settings, contact lists, photo albums, text documents, and many more. This form of caching is already prevalent in the industry (e.g., OneDrive, Dropbox), but has not been standardized to a broad platform such as dew computing.

Personal data which the user cannot modify:

- It is certainly possible that one user can access personal data of another user, without the ability to modify it. In this scenario, we treat the data in the same way as public and group read-only data - always read locally and tolerate slightly out-of-date data in a weakly-consistent fashion. The latest changes are guaranteed to be synchronized at some point in the future.
- Perfect examples of personal data which the user cannot modify are friend feeds in social media sites such as Facebook or Instagram.

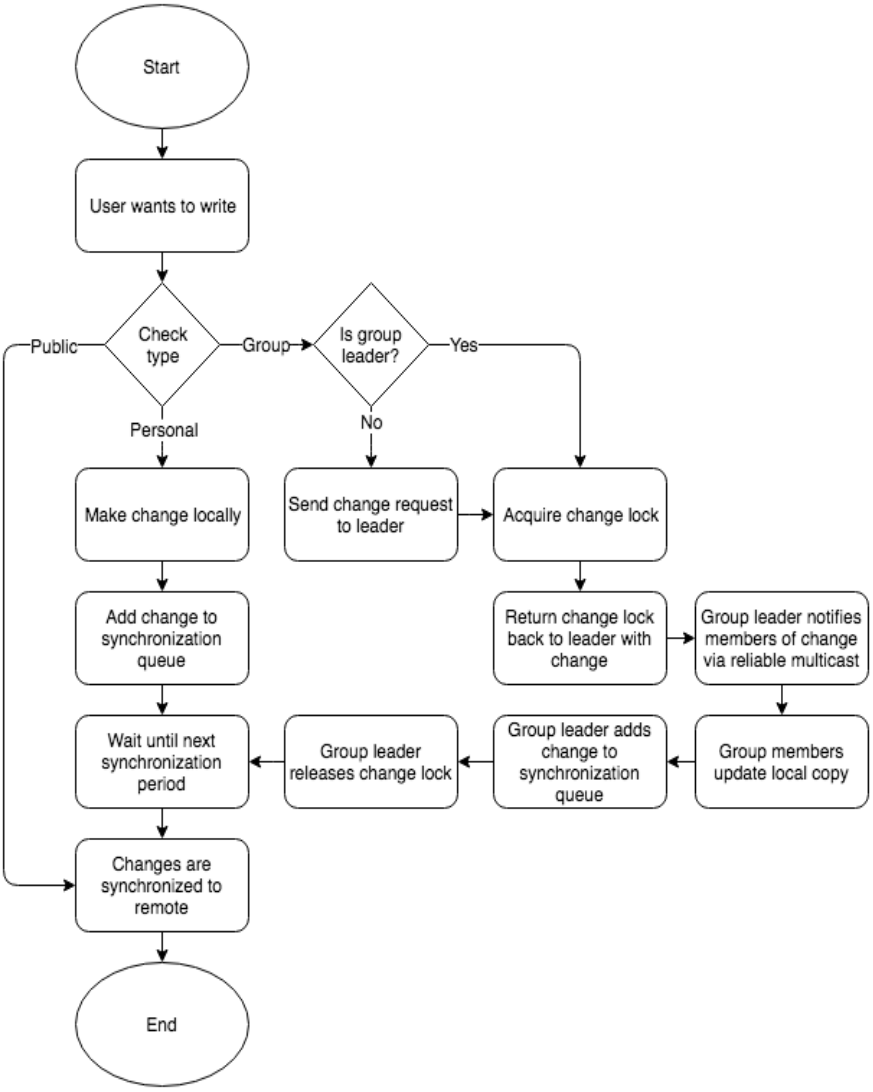

Figure 5. Flowchart comparison of local write operations

4.3. Synchronization

Synchronization plays an important role in maintaining the freshness and stability of the overall system. When a client machine modifies some data or state in their local environment, the changes are not required to be immediately synchronized to the remote server. Instead, the client machine adds the changes to what is known as the synchronization queue. As time goes on, more and more changes will get added to the queue, each timestamped with the time the change was made.

Concurrent with this client process, the remote server is performing a similar activity. Every change that some other client machine has made will eventually be reflected in the remote cloud copy, and the cloud servers maintain a separate queue containing changes relevant to each user. For example, if Client 1 has cached some personal data of Client 2 and if Client 2 makes some changes to their personal data which is later synchronized to the cloud, then the cloud adds Client 2's changes to Client 1's synchronization queue. After some predefined interval, a synchronization event will be triggered by the client machine.

The purpose of this synchronization event is to exchange the changes stored in the client-side synchronization queue with the changes stored in the server-side synchronization queue and vice-versa. After the synchronization event has completed, the state of the client machine and the remote machine will temporarily be in-sync - at least until more changes get added to one or both of the synchronization queues. Figure 6 visualizes the high-level synchronization process below.

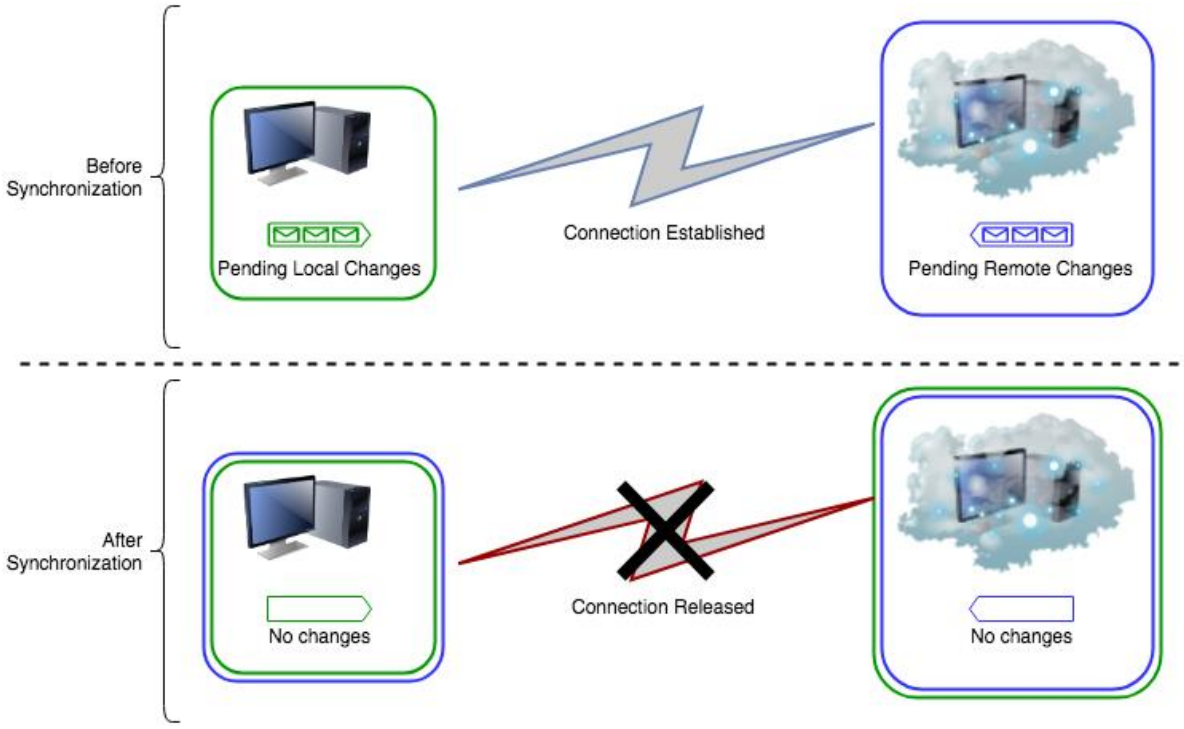

Figure 6. Synchronization process

Note that there is a time delay in the synchronization process; it does not continuously run. The benefit to this approach is that multiple changes can be clubbed together into a single event, reducing the load on the network. The downside to this approach is that both the client and server can never be considered 100% synchronized and the client may notice a small delay before they have access to the latest data locally. We remind our readers that the lack of complete synchronization is an intentional design decision to increase the availability and scalability of the system, and it does not impact the consistency of writable data. Writable data will always be strongly consistent, as defined within the caching policies in Section 4.2.2.

5. Summary & Future Work

Dew computing is still a budding new technology with a lot of room for improvement. Dew was created to further the idea of the fog architecture to put more data closer to the user. Just like all new innovations, dew still has some limitations to overcome.

Our proposed solutions for the remote server-side (Cloud) architecture include: the data segregation layer, operation segregation layer, and container creation and deployment. Our proposed solutions for the client-side (local) architecture include: leveraging localized data and functionality, data modification, and cache consistency. We encourage the readers to brainstorm their own solutions. After all, progress is a collaborative effort that is never ending.

Future work could include implementing some of the ideas proposed in this paper and benchmarking how they perform against the traditional cloud computing architecture. Developing a small scale service like a mini social-network that follows the dew computing paradigm would be a great start.

6. References

[1] Wang, Yingwei. "Definition and categorization of dew computing." *Open Journal of Cloud Computing (OJCC)* 3.1 (2016): 1-7.

[2] Šojat, Zorislav, and Karolj Skala. "Views on the role and importance of dew computing in the service and control technology." *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016 39th International Convention on*. IEEE, 2016.

[3] Gusev, Marjan. "A dew computing solution for IoT streaming devices." *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2017 40th International Convention on*. IEEE, 2017.

[4] Ristov, Sasko, Kiril Cvetkov, and Marjan Gusev. "Implementation of a horizontal scalable balancer for dew computing services." *Scalable Computing: Practice and Experience* 17.2 (2016): 79-90.

[5] Sasikala, P. (2011) 'Cloud computing: present status and future implications', *International Journal of Cloud Computing*, Vol. 1, No. 1, pp.23–36.

[6] Rimal, B.P., Choi, E. and Lumb, I. (2009) 'A taxonomy and survey of cloud computing systems', *in NCM '09: Proceedings of Fifth International Joint Conference on NC, IMS and IDC*, Seoul, Korea, pp.44–51.

[7] Wang, Y. (2015) "Cloud-dew architecture", *Int. J. Cloud Computing*, Vol. 4, No. 3, p 199-210.

[8] Wang Yingwei, Karolj Skala, Andy Rindos, Marjan Gusev, Yang Shuhui, and Pan Yi, "Dew Computing and Transition of Internet Computing Paradigms"

[9] Zorislav Šojat, Karolj Skala, "Views on the Role and Importance of Dew Computing in the Service and Control Technology"

[10] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Harlow, England: Addison-Wesley, 2012.

[11] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015.

[12] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java® Virtual Machine Specification Java SE 8 Edition*. Upper Saddle River, NJ: Addison-Wesley, 2014.

[13] A. Kahol, S. Khurana, S. K. S. Gupta, and P. K. Srimani, "A strategy to manage cache consistency in a disconnected distributed environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 7, pp. 686–700, Jul. 2001.

[14] M. Blaze, "Caching in Large-Scale Distributed File Systems," PhD thesis, P*rinceton University*, Jan. 1993.

[15] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell, "A hierarchical internet object cache" in *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, Jan. 1996.

[16] C. Liu and P. Cao, "Maintaining Strong Cache Consistency in the World-Wide Web," *Proceedings of 17th International Conference on Distributed Computing Systems*, pp. 530–537, 2000.

[17] Wikipedia, “1% rule (Internet culture),” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/1%25_rule_(Internet_culture). [Accessed: 29-Apr-2018].
[18] T. V. Mierlo, “The 1% Rule in Four Digital Health Social Networks: An Observational Study,” *Journal of Medical Internet Research*, vol. 16, no. 2, Feb. 2014.