Data Management in Distributed Systems

Simon Schäffner
Advisor: Stefan Liebald
Seminar Innovative Internet Technologies and Mobile Communications SS2018
Chair of Network Architectures and Services
Department of Informatics, Technical University of Munich
Email: simon.schaeffner@tum.de

ABSTRACT
Distributed Systems allow a number of nodes to collaborate on a problem. They can be of help when a single server can no longer fulfill the requirements or when a larger number of low performance nodes would like to cooperate. This paper gives a broad overview over different strategies for data management in different kinds of distributed systems. It focuses on the issues of scalability, performance, consistency, redundancy, overhead and attack resistance. Five systems from three categories are reviewed: peer-to-peer file sharing (BitTorrent and Kademlia), content delivery networks (Akamai) and distributed databases (Zatara and CouchDB).

Keywords
Distributed Systems, Data Management, Peer-To-Peer, Content Delivery Network, Scalability, Performance, Consistency, Redundancy, Overhead

1. INTRODUCTION
Since the beginning of computing, there have been two important developments: the advancement from single-core to multi-core processing and the establishment of local and global high-speed networks. This allows for a variety of different kinds of computers, ranging from small credit-card sized computers or even smartphones to supercomputers, to connect together and form a distributed system [19]. This system can be characterized as "a collection of autonomous computing elements that appears to its users as a single coherent system" [19]. "autonomous" refers to the "computing elements", called nodes, all being independent of each other and not being controlled by a central instance. "appear[ing] as a single coherent system" means that the system has a larger goal and therefor nodes have to collaborate to achieve the goal. In addition to these two characteristics, it allows for the nodes of a distributed system to be geographically dispersed.

Most tasks either require some data input, an ability to output/save data or even both. This is also the case in distributed systems, but data management is usually a lot more complex than just reading and/or writing data to a single disk.

This paper aims to compare data management strategies in different distributed systems of different categories. Mostly, popular systems with interesting and/or unique features were chosen. The goal is not to give an in-depth review of the strategies, but rather to give a broad overview over strategies that are actively used at the time of writing.

In the following, first, different attributes of data management systems are defined, and then the systems are compared based on these metrics. In section 2 we define different attributes that data management in distributed systems can have. In section 3 we take a look at 5 concrete distributed systems and compare their data management by the attributes defined in section 2. In the summary we compare all the systems and give an overview of them in table 1.

2. ATTRIBUTES
We now describe the six attributes the distributed systems are then compared by. The attributes were chosen as they are relevant for most systems building on top of them.

2.1 Scalability
Because of the openness of the internet, a service provider never knows when a spike in the popularity of their service might happen. Therefor they would like to be able to scale their applications indefinitely, as long as it makes sense economically. With the whole system, the data management system has to be able to scale as well.

2.2 Performance
Performance in this context describes how fast a requested datum can be accessed. This includes both the initial delay to find where the datum is stored and how fast it can then be transferred to the requesting node.

2.3 Consistency
For redundancy a datum may be stored on more than one node. This implies a challenge whenever a datum is changed as the change has to be distributed through the system. Consistency describes how well this is handled and if there is the chance of receiving stale data.

2.4 Redundancy
As described above, a system can store data in more than one place to ensure its availability or to increase performance. Higher redundancy is better.

2.5 Overhead
Because of the nature of distributed systems, nodes inside them have to communicate one way or another. One extreme is that all nodes have all data saved in their local storage,

doi: 10.2313/NET-2018-11-1_15

but then any change in data has to be broadcast. The other extreme is that the data is completely distributed between the nodes, but then the nodes have to request data from each other. Independent of the strategy for redundancy, this communication obviously needs additional resources called overhead over a single computer directly accessing data on its internal disk. Less overhead is better.

2.6 Attack Resistance

The open nature of the internet implies that distributed systems may be attacked. In the case of data management, an attack could mean not being able to deliver data to the rest of the system anymore or delivering manipulated data. Attack resistance describes how many attack vectors there are and how significant each is.

3. COMPARISON

In the following we are going to compare different distributed systems, that have a focus on data management. We chose two protocols relevant for peer-to-peer filesharing, BitTorrent and Kademlia, the content delivery network Akamai and two distributed databases, Zatara and CouchDB.

3.1 Peer To Peer Filesharing

While peer to peer filesharing first emerged with the development of Napster[1] and gnutella[2] for legally controversial uses, it has advantages over traditional server-client structure downloading, that makes it interesting for completely legal use, as well [18]. It uses shared resources to split the load between many participants and has a higher reliability as there are fewer single points of failure.

3.1.1 BitTorrent

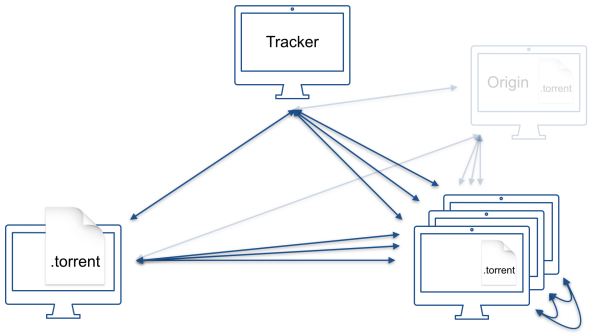

Figure 1: BitTorrent network with offline origin

BitTorrent is a peer-to-peer file sharing protocol specified in [10]. It is still under active development, and is still the most popular peer-to-peer file sharing protocol on the internet, but its usage falls with streaming services, such as Netflix[3], gaining more popularity. [7]

A download is started by first downloading a .torrent file containing all the meta information needed. This includes the URI of a so-called tracker, which coordinates all the peers downloading the same file. A peer, also called a leecher when it has not yet completed the download, connecting to a tracker receives a list of other peers, which have parts of the file available that the leecher still needs. As long as the network as a whole has the complete file available, every peer can download the needed parts from the other peers. Initially one peer, the 'origin', has to have the whole file available at once.

The intention of a BitTorrent network is to distribute one or more files to anyone or any node that can access the .torrent file and connect to the tracker. So the goal of this distributed system is data management itself. On a higher level, metadata also has to be managed. Especially the .torrent file has to be managed externally, so it is not considered here.

In the following BitTorrent is analyzed according to the attributes described in section 2.

Scalability: According to a two-week experiment described in [16], the download speed scales with the number of peers. The trackers are no bottleneck as they only have to distribute metadata. The more problematic issue is fairness. An investigation[21] shows that peers with a high download speed from the origin become "Super Peers" that have a very high share ratio (upload to download ratio). In all five repetitions of the experiment the group of "Super Peers" consisted of the same peers independent of the time they joined the download. This is unfair because a small number of peers has to provide a larger amount of resources than the other peers. Independent of fairness, scalability is still good.

Performance: BitTorrents upload utilization is very good as reported in [6]. In their experiments in a homogeneous environment (same bandwidth for all leechers) the upload utilization was around 95%, independent of the number of peers. The download utilization was lower because it was bounded by the upload speed of the leechers, so the bottleneck was the upload bandwidth, not the protocol being incapable. Performance of BitTorrent is very good because of the upload utilization.

Consistency: As the content of a torrent never changes and the user first downloads the metadata file containing checksums of all pieces of the file(s), consistency is not an issue with BitTorrent. In the worst case scenario of downloading a corrupted piece, the piece is just discarded and downloaded again. In the case of downloading a corrupted metadata file, the BitTorrent client will notice a difference in the checksum from the metadata file and the checksum provided by the tracker and will stop the download [10].

Redundancy: As with all file-sharing platforms, a higher redundancy is better, because it infers a higher availability. There is also no disadvantage of high disk space utilization as every peer who downloads the file also wants to use it, so they want to have it available locally to them. With BitTorrent, a higher redundancy also implies a higher download speed for new leechers.

Overhead: The obvious overhead of BitTorrent, compared to downloading a file from a single source, is the metadata

[1] `https://us.napster.com`
[2] `http://www.gnutellaforums.com`
[3] `http://netflix.com`

doi: 10.2313/NET-2018-11-1_15

file, the connection to the tracker and the meta information sent when connecting to other peers. The connection to the tracker makes up for a thousandth of all traffic, which makes it negligible [9]. The already mentioned overhead of replicating the file to every node in the system and thereby using a lot of disk space is not negative as this is the goal of the system.

Attack Resistance: While there are few attack vectors on BitTorrent that actually result in the successful download of a corrupt file, there are methods to hinder a download. These are called poisoning attacks and are executed by anti-piracy-agencies and malicious users. Poisoning attacks include uploading a large amount of fake files and/or malware, so that users cannot find the file they are looking for with a search engine [11]. Another possible attack is flooding all peers that offer parts of the file with download requests so that it cannot be downloaded by others. This and other methods are even offered as a service by companies like MediaDefender [5].

All in all, BitTorrent is a peer-to-peer file sharing protocol with high performance, a good strategy for consistency, little overhead and a small number of attack vectors.

3.1.2 Kademlia

Kademlia [14] is a peer-to-peer distributed hash table (DHT). Even though Kademlia was developed as a research project, it has found its way into practical usage with it being integrated into BitTorrent [13] and the cryptocurrency Ethereum [20].

The identifier space is used for both keys and node IDs. Keys are stored on nodes whose IDs are "close". Because the whole network is interpreted as a binary tree, the magnitude of the distance in a fully populated tree is determined by the height of the smallest subtree the two nodes are part of. The path from the root of the network to the root of the subtree sets the prefix for all identifiers in the subtree.

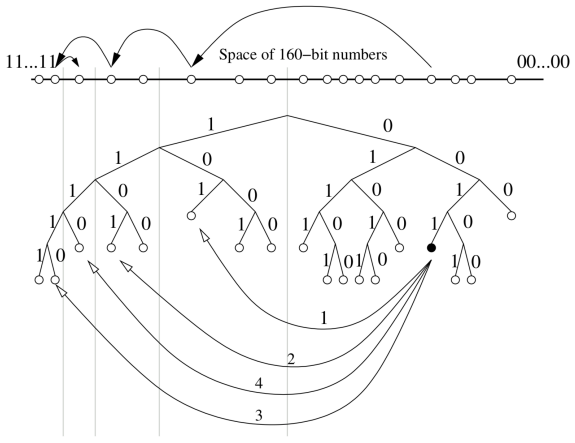

Figure 2: Finding a key in Kademlia [14]

To ensure that a node can find every other node in the network, it has to know at least one node from every subtree it is not part of. To look up the value for a given key, the node sends parallel requests to the k nodes it has contact with closest to the key. Each of the nodes returns the k nodes *it* has contact with closest to the key. The requesting node then sorts the nodes by distance to the key and contacts the k closest nodes. This is done recursively until one of the contacted nodes returns the value.

To store a <key,value> pair, a node searches for the k closest nodes to the key and sends them a store command. When a new node joins the network, it introduces itself to the system and stores all <key,value> pairs it is one of the k closest nodes to.

Kademlia can be used as a way to remove nearly all central components from a peer-to-peer network. In this context it provides a way to find data and information on how to access it, so in the case of BitTorrent it replaces the central trackers [13].

Scalability: The amount of storage scales linearly with the amount of nodes as the redundancy-factor k is global and not dependent on the amount of nodes in the network. Of course, with an increasing amount of nodes in the network, the amount of nodes that have to be contacted to find the value of a key increases, but because of some optimizations in the tree lookup system this scales even better than $O(log_2(n))$ [14] which is good.

Performance: Kademlia uses parallel, redundant requests to decrease performance-loss by broken nodes or nodes that have left the network. When another node does not reply within a certain time, the node is removed from the requesting node's contact list and is no longer contacted until it reintroduces itself. The system also uses caching to decrease the likelihood of creating a hotspot on a single node when a certain <key,value> pair is often requested. Whenever a node requests a value for a given key, it sends a store command to the node closest to the key that did not have the value stored before. Because of the unidirectional topology, requests from other nodes for the same key are highly likely to hit the caching nodes. With parallel requests and optimized caching Kademlia's performance is very good.

Consistency: The original publisher of a <key,value> pair has to republish it every 24 hours to limit stale information in the system. Each of the k nodes the <key,value> pair is stored on primarily has to republish it every hour. To decrease the amount of cached stale information, the time to live in the cache is exponentially inversely proportional to the distance between the node the <key,value> pair is primarily stored on and the node it is cached on. Since stale information can live a long time in Kademlia, its consistency is not outstanding.

Redundancy: <key,value> pairs are stored at least k times in the system as they are stored on the k closest nodes to the key, so whenever a node leaves the network (or dies) at least $k-1$ nodes should still have the information available. Due to republishing, after a maximum of one hour, the information should be available on k nodes again. When all k nodes that store the same <key,value> pair leave the network in a single hour, the information should be available again after 24 hours at last. With k chosen to have a small likelihood

doi: 10.2313/NET-2018-11-1_15

of k nodes leaving the network in one hour, Kademlia's redundancy is good.

Overhead: As already mentioned, <key,value> pairs have to be republished every hour. This process is optimized by a node not republishing for the next hour when a republish was received, as it is assumed that the other nodes have received the republish as well. Caching also introduces some overhead but that is limited to short time periods and to <key,value> pairs that are requested often. It also decreases the amount of hops needed to receive a value for a given key, so it reduces network traffic which is the more precious resource in a peer-to-peer system that runs on end users' home computers.

Attack Resistance: One attack vector for open distributed systems is to overtake a large part of the system and by that being able to control the whole network. This is difficult with Kademlia as the existing nodes do not replace nodes in their contact list that have been longer known to them with newer ones. One reason for this is that statistically, nodes that have been in the network for a longer time have a smaller risk of leaving the network. Another reason is that a malicious attacker cannot overtake the network by flooding it with new nodes.

Overall Kademlia has a good scalability, a great performance, but falls short on consistency and overhead as keys have to be actively republished but cannot be actively removed.

3.2 Content Delivery Networks

Websites usually start out running on a single server, but when they grow large, even a single cluster of servers is no longer enough. One possibility is to use proxies in front of the content generating servers in order to cache static content. But now that a large amount of content on the internet is generated dynamically, hit rates of proxies are low (25-40%) [12] and more elaborate systems called Content Delivery Networks (CDNs), were developed. These systems consist of a large amount of servers distributed both geographically and regarding network topology. Whenever a user wants to access a website, the user actually connects to one of the CDN's servers instead of the actual website's servers. The CDN only connects to the content generating server if it has to.

3.2.1 Akamai

Akamai is one of the world's largest content delivery networks (CDN) with well known customers such as Facebook, Adobe and Airbnb. They have "more than 240,000 servers in over 130 countries and within more than 1,700 networks around the world"[4].

When a large amount of users hits a single website at once (a flashcrowd), Akamai allocate more of their servers to the website that needs them at the moment and less to others. They also try to serve users from a server nearby, so that latency is low and packet-loss is small [12].

For customers with large amounts of data, Akamai use tiered distribution within their own network: A set of "parent" clusters (see figure 3) is used to cache the data within the network, and when another cluster does not have that data available, it retrieves it from the parent cluster instead of the origin. This can result in offload of over 90% [15].

Akamai also handle livestreams through their network [15], but this is out of the scope of this paper.

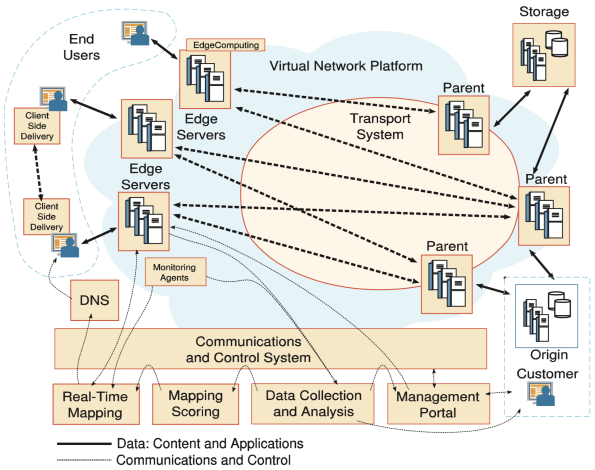

Figure 3: Overview of the Akamai network [15]

Scalability: Akamai was designed with scalability in mind [15]. Tiered distribution allows for a large amount of data being available within the network and therefor being available at high speed to the whole overlay network.

Performance: Performance was another goal Akamai was designed for [15]. They have developed their own overlay network with a number of improvements over using standard internet. One advantage is path optimization, as paths defined by the Border Gateway Protocol are not always optimal and sending traffic through an intermediate server on the Akamai network can be faster. This can show improvements in speed of 30-50% [17]. On top of that it can increase reliability by offering alternate paths. Another advantage of path optimization is reduced packet loss. For applications requiring low latency, a packet can be sent via multiple ways simultaneously, which decreases the chance of a packet not reaching its destination at all. This is also combined with forward error correction techniques to decrease the amount of times packets have to be discarded because of transmission errors. Additionally, Akamai uses transport protocol optimizations to mitigate the overhead of protocols like TCP. Further, application level optimization is used when possible. This includes content compression or even the implementation of application logic on edge servers [15].

Consistency: Data management for cacheable objects is mostly based on standard techniques such as assigning time-to-live values to each object or using different URLs for different versions of the same object. But as Akamai only serve content for their customers, they expect to retain control over their data [15].

Redundancy: How often a datum is stored in the Akamai network depends on the customer and their needs. As mentioned above, for a customer Akamai is an extension of their

doi: 10.2313/NET-2018-11-1_15

own network and the amount of redundancy is completely dependent on their application. With tiered distribution Akamai tries to strike a balance between redundancy and fast availability within their own network.

Overhead: As Akamai's overlay network is proprietary, the protocol overhead is not known, but they claim it to have improvements over standard TCP as mentioned above (see performance). This is done by reducing the amount of times a connection has to be setup and torn down, as they can leave up connections within their network for a longer time.

Attack Resistance: As Akamai is a closed network, there are no attackers within the network that could try to destroy data or deliver wrong data. Attacks from the outside are still possible, but Akamai was engineered with a high failure rate of equipment and connections in mind. Because of that, a lot of effort was put into recovery from all kinds of failure scenarios [15].

All in all, the highly distributed nature is fundamental to Akamai's high performance. The entire communication within the overlay network is optimized and the two small hops on either end are meant to be short enough that they do not matter. This means great scalability, performance and redundancy.

3.3 Distributed Databases

A distributed database can partition data over many servers, for OLTP use-cases (transactional processing: a large amount of simple queries) each query can be handled by a different node or for OLAP use-cases (analytical processing: a small amount of complicated queries) the nodes can work together on a single query. With replication, they can also make large amounts of data available to geographically dispersed systems with a low speed link in-between.

Zatara was chosen as it was one of the first general use-case NoSQL databases. CouchDB was selected for its ability to gracefully resynchronize a client that was offline for some time.

3.3.1 Zatara

Zatara is an eventually consistent distributed database built to satisfy the needs of modern cloud applications [8]. Opposite to relational databases like MySQL or Postgresql, Zatara is a NoSQL database. NoSQL databases allow for other data models than tables and support other query systems than SQL. Zatara, in particular, is more similar to a key-value-storage.

Each key can be of the type cache only or persistent storage. Cache only keys are stored on a single node in memory and are not replicated at all. This implies that they are lost on node failure and are non persistent. Persistent keys are eventually consistent and are stored on disk. They are also replicated to all other nodes in the group.

Each key to be stored is mapped to a single node by a hashing algorithm and the client connects directly to that node to store the key. For cache only keys this is the one node that stores the key and for persistent keys this is the primary node the key is stored on and the node that handles replication for this key.

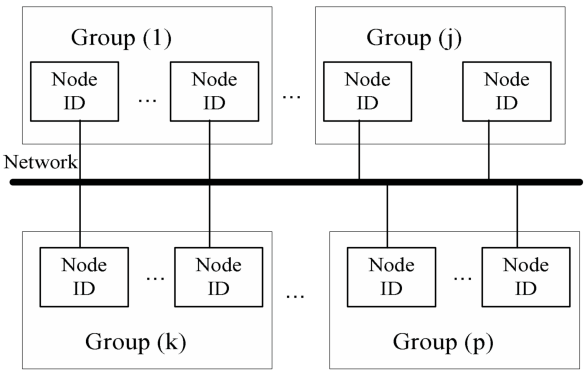

Figure 4: Zatara nodes organized in groups [8]

When a new node joins the network, there are two options: The first option is that the node joins an existing group and replicates the keys that are stored in the group. The other option is that a group is split. For this, the keys of the old group have to be rehashed and are thereby divided between the two new groups.

Every node is configured with a unique node id and authentication information. To connect to the network, each node is given a list of other nodes in the network.

Scalability: Data is replicated regionally only, whereby replication data only has to be sent to a small amount of nodes, instead of the whole network. Data is also replicated asynchronously after it has been stored on at least a second node, which brings a good scalability.

Performance: One of the performance improving design aspects of Zatara is that keys are cached in memory and the more often they are requested, the higher a TTL they are being assigned. This decreases the chance of the key being deleted in case the node runs out of memory. In more practical terms, the authors not only describe Zatara in [8], but also evaluate its performance using 196 Amazon EC2 Large Instances as nodes [8]. They come to the conclusion that performance scales near linearly without any replication and is only 10% slower with replication for reading a key from the database. In [8] they note that they were "able to reach more than 20 million operations per second in a pay-as-you-grow cloud infrastructure that costs less than 100USD per hour."

Consistency: Zatara offers eventual consistency[4], so it guarantees that the results for a changed entry will be consistent after the consistency window has closed. This mechanism is implemented by the node that primarily stores the key actively distributing the new value for the key to the other nodes in the group. As soon as at least one other node of the group has acknowledged the update request, the node acknowledges the request to the client. By this, Zatara

[4]consistency is guaranteed only a given timespan after the data update is completed, not directly, as it is with traditional database systems

doi: 10.2313/NET-2018-11-1_15

can also guarantee a persistent key to be always stored on at least two nodes. If a node cannot be contacted, the sending node will increase a soft fail counter for that node. If the soft fail counter increases above a certain threshold, the node's status will be set to HardFail and other nodes will be informed.

Redundancy: Persistent keys in Zatara are guaranteed to be stored on at least two nodes. When the primary node for the key is not available, the key will be read from another node in the group. As long as the groups are large enough, redundancy in Zatara is good.

Overhead: Asynchronous replication is used to reduce overhead compared to synchronous replication, but it is still a problem in large networks with many nodes. To solve this problem, nodes are organized in groups with recommended group sizes of 2-4 nodes per group. With the recommended group size, Zatara can strike a balance between redundancy and overhead.

Attack Resistance: Zatara employs an internal key to store username and passwords and another internal key to store which users are allowed access to which databases. Every client and every node first has to authenticate against the internal key, so in theory only trusted clients and nodes should have access to the network.

All in all, Zatara is a distributed database with very good scalability because of asynchronous data replication, good performance because of in-memory caching and eventual consistency. Overhead and attack resistance are neither outstandingly good nor bad.

3.3.2 CouchDB

CouchDB [3] is a document storage NoSQL database aimed at web applications. Documents are the main unit of data, consisting of a number of fields and attachments [1].

One of the primary use cases of CouchDB is to have multiple offline nodes and then synchronize the data upon reconnect. CouchDB is designed to handle partitioning of the system, especially when a single node disconnects and reconnects to the network, gracefully [1].

Internally, documents are versioned and when a new version of a document is saved, it is appended to the end of the database file. Occasionally, all current versions are copied into a new file and the old file is deleted when no clients use it anymore [1].

Scalability: CouchDB restricts data lookup to keys, which allows for data to be partitioned between many nodes without loosing the ability to query each node individually [1]. This brings good scalability.

Performance: Locally (on a node) data is stored in B-trees, so that data lookup by key efficient ($O(logN)$). Additionally, CouchDB uses multiversion concurrency control (MVCC) instead of locks [1]. With MVCC, queries see the state of the database at the beginning of the query for their whole lifetime. Because of this, queries can be run fully in parallel as long as they do not write to the same data set. As already mentioned, data can be partitioned between nodes to handle large amounts of data and large queries. Because of efficient data lookup and parallel requests, performance of CouchDB is good.

Consistency: A single CouchDB node is fully ACID (availability, consistency, integrity, durability) compliant. As mentioned above, MVCC is used for reading queries. When a client tries to save a document that in the meantime has been edited by another client, an edit conflict is triggered. The client then is offered the option to resolve the conflict by reapplying the changes to the newer version of the document. A conflict is also triggered, when an offline node reconnects to the network and a document was changed on both the network and the node. Each node deterministically decides which version of a document wins in this situation (usually the newest one). The loosing versions are still stored in the database and can still be accessed. They are only purged on database compaction. Loosing versions, like any others, are replicated among the network. Because of that, every node in the network sees the conflict and has the option to resolve it either automatically or manually [1]. This is an elaborate system, similar to version control systems like git[5].

Redundancy: This is left for the user of the database system to decide upon. Databases can only run on a single node, can be fully replicated between many nodes, can be partitioned between many nodes or any configuration in between. All of these options have their own advantages and disadvantages resulting in higher or lower query speed and higher or lower availability [1].

Overhead: As mentioned above, data can be fully replicated between a number of nodes in the system. This obviously brings a large overhead in both storage and communication. When a node, that was offline for a while, reconnects to the network, all changes have to be transferred to it and all the changes on the reconnecting node have to be transferred to other nodes in the network. The amount of overhead depends on the level of redundancy, so it is not a negative.

Attack Resistance: For external security, CouchDB implements a simple authentication model by default, but also allows for a custom, more complex model. One can implement a JavaScript function for update validation that receives both the new state of the document and authentication of the client to decide on that basis if the update should be allowed [1].

All in all, CouchDB is a very flexible system being both useful with data partitioned over many servers and a single instance running on a smartphone. It offers very good performance and an elaborate system for consistency.

4. SUMMARY

In table 1 an overview of all systems and their particular advantages and disadvantages is given. All of the systems are marked at least "good" in performance, as every system is optimized for at least one use-case and performs well in it.

[5] `https://git-scm.com`

doi: 10.2313/NET-2018-11-1_15

Table 1: Overview of All Systems

System	Data update	Scalability	Performance	Consistency	Redundancy	Overhead	Attack Resistance
BitTorrent	-	++ dl. speed scales lin. with #nodes	+ upload util. very good	0 no data updates; corrupt data identified	+ high	0 high, but aligns with goals	- poisoning attacks possible
Kademlia	active republication	+ storage scales lin. with #nodes; lookup time scales with $O(log_2 n)$	++ parallel redundant requests, efficient caching	+/0 old information max. 24h in system	+ $< key, value >$ pairs stored on $>= k$ nodes	- optimized process republishing $< key, value >$ pairs every hour	+ cannot be overtaken by node flooding
Akamai	dep. on customer	++ >240.000 nodes	++ optimized transfer speed in network	0 based on TTL/version URLs, but dep. on customer	++ tiered distr. allows for any degree	0 improvements over standard TCP, dep. on redundancy	++ only attacks from outside possible; high recovery rate
Zatara	active	++ asynchr. data repl.; large #groups possible; $O(1)$ key lookup	++ in-memory caching (least recently used)	+ eventual consistency	+ keys are guaranteed to be stored on >= 2 nodes	0 regional replication	0 simple authentication
CouchDB	active	++ data lookup by key only; data can be partitioned over many nodes	++ key lookup locally $O(logN)$; MVCC for parallel reading queries	++ eventual consistency; graceful conflict handling	++ dep. on use-case	+ dep. on redundancy needed	+ simple auth. model by default, can be customized

There are also interesting differences between the systems. BitTorrent does not allow for the actual data to change and is the only system that uses passive data update (for metadata) by having the nodes poll from the tracker. All other systems use active updates for replication. With Kademlia data is republished every hour, Zatara actively updates the values during the replication window and CouchDB replicates data actively on change or on reconnection to the network. Akamai with tiered distribution can invalidate cached data actively, dependent on the customer's needs.

It makes sense that active replication was found the most as it decreases the time of the network being inconsistent. It brings more overhead if only a single node needs the data in the end, but that does not align with the goal of high availability that most systems have.

There are also different strategies for deciding on which nodes to store data. BitTorrent stores all data on all nodes. Kademlia stores a key on the k closest nodes to the key. With Zatara a key is stored on a primary node that then replicates it to the rest of the group. CouchDB is very flexible on this issue as one can use a custom filter function for every node to apply during replication. With Akamai using tiered distribution, data is stored on a few primary nodes first and is then cached by other nodes when it is needed. Which nodes are chosen as the primary ones is dependent on the customer's needs, e.g. where the customer's main userbase is from geographically.

There are also different ways for conflict handling. CouchDB is the only system that has an elaborate strategy. As mentioned above, it sets the key to a conflict state and then lets the clients handle it. All the other systems do not do conflict management. BitTorrent does not need it as data cannot be changed. Tiered distribution also does not create any conflicts as data comes from a single place and is propagated from the primary nodes. A node in a Kademlia network looses all data on disconnect and therefor cannot create a conflict on reconnect. With Zatara all writes are directed to the same node and are then replicated from there, so there is a single authority for every key in the system. Larger partitions breaking off the network and then rejoining it is not provided for.

5. CONCLUSION

All of the systems, with the exception of Zatara, are popular at the time of writing: BitTorrent is the most popular peer-to-peer filesharing protocol [7], Kademlia is implemented in BitTorrent [13], Akamai reaches more than 30 Terabits per second of traffic and CouchDB is used by many small companies and even large ones like Akamai [2].

A short overview of the different protocols used in the systems was given and their strategies for data management were compared. Each of them has their own advantages and disadvantages.

As mentioned, distributed systems could only be developed because of high-speed networks. The worlds largest high-speed network - the internet - is available and here to stay, so the amount of distributed systems will only ever increase. With more and more aspects of peoples lives moving online, systems will have to be employed that support a very large workload and can be accessed from all over the world. The only type of systems available to handle this, are distributed systems.

6. REFERENCES

doi: 10.2313/NET-2018-11-1_15

[1] 1. introduction. http://docs.couchdb.org/en/2.1.1/intro/index.html. last accessed: 02.06.2018 15:09 (revision f85b3421).

[2] Companies using couchdb. https://idatalabs.com/tech/products/couchdb. last accessed: 02.06.2018 15:11.

[3] Couchdb - relax. http://couchdb.apache.org. last accessed: 02.06.2018 15:10.

[4] Facts & figures. https://www.akamai.com/us/en/about/facts-figures.jsp. last accessed: 27.05.2018 20:47.

[5] N. Anderson. Peer-to-peer poisoners: A tour of mediadefender. https://arstechnica.com/tech-policy/2007/03/mediadefender, May 2017. last accessed: 27.05.2018 15:38.

[6] A. Bharambe, C. Herley, and V. N. Padmanabhan. Analyzing and improving bittorrent performance. 01 2006.

[7] K. Bode. Netflix now accounts for 36.5% of peak internet traffic. http://www.dslreports.com/shownews/Netflix-Now-Accounts-for-365-of-Peak-Internet-Traffic-133945, May 2015. last accessed: 02.06.2018 15:12.

[8] B. Carstoiu and D. Carstoiu. High performance eventually consistent distributed database zatara. In *INC2010: 6th International Conference on Networked Computing*, pages 1–6, May 2010.

[9] B. Cohen. Incentives build robustness in bittorrent. http://www.bittorrent.org/bittorrentecon.pdf, May 2003. last accessed: 27.05.2018 14:51.

[10] B. Cohen. The bittorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html, Feb 2017. last accessed: 18.05.2018 18:44.

[11] R. Cuevas, M. Kryczka, Á. Cuevas, S. Kaune, C. Guerrero, and R. Rejaie. Is content publishing in bittorrent altruistic or profit-driven? In *Proceedings of the 6th International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT 2010)*, http://hdl.handle.net/10016/10116, December 2010.

[12] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, Sep 2002.

[13] A. Loewenstern and A. Norberg. Dht protocol. http://www.bittorrent.org/beps/bep_0005.html, Jan 2008. last accessed: 02.06.2018 15:11.

[14] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, pages 53–65, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[15] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, Aug. 2010.

[16] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In M. Castro and R. van Renesse, editors, *Peer-to-Peer Systems IV*, pages 205–216, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[17] H. Rahul, M. Kasbekar, R. Sitaraman, and A. Berger. Towards realizing the performance and availability benefits of a global overlay network. *MIT CSAIL TR 2005-070*, Dec. 2005.

[18] R. Steinmetz and K. Wehrle. *2. What Is This "Peer-to-Peer" About?*, pages 9–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[19] M. van Steen and A. S. Tanenbaum. A brief introduction to distributed systems. *Computing*, 98(10):967–1009, Oct 2016.

[20] vbuterin and J. Ray. Kademlia peer selection. https://github.com/ethereum/wiki/wiki/Kademlia-Peer-Selection, Oct 2015. last accessed: 02.06.2018 15:13 (revision ea47c31).

[21] Z. Zhang, Y. Li, Y. Chen, P. Cao, B. Deng, and X. Li. Understand the unfairness of bittorrent. 12 2010.