DDD: Distributed Dataset DNS

Joseph White-Swift¹ · Andrea Fumagalli¹

Received: 29 April 2021 / Revised: 14 October 2021 / Accepted: 6 December 2021 / Published online: 4 February 2022
© The Author(s) 2022

Abstract

The advent of inexpensive data storage has resulted in larger and larger datasets as the cost of pruning data becomes more expensive than storing it for future insights. This decreasing cost of storage has also led to the practice of storing data in multiple locations for redundancy. However, without any uniform method of determining link costs to different storage sites, a dataset is not always retrieved from the most cost effective site. Distributed dataset DNS, or DDD, solves this problem in two key ways. The first allows “local” servers to provide meaningful information to a user in order to ensure that they target the location that offers the most advantageous network connection. The second allows other trusted servers to easily gain access to this information in a distributed way. These combined approaches aim to both lower aggregate network bandwidth usage and prevent single points of failure when retrieving dataset pointers.

Keywords Distributed dataset DNS · Efficient retrieval of datasets · Decentralized peering · Protocol

1 Introduction

The demand for a combined solution that can lower aggregate bandwidth usage and decrease download times simultaneously will continue to grow as the amount of data produced and stored globally increases. While each of these inefficiencies is small, the combined effect is real and costly. While some protocols do bits and pieces to solve this problem, no protocol exists that positively affects users, administrators, and network operators while working in a distributed, decentralized way. Therefore, the distributed dataset DNS (DDD) protocol is presented as a solution to a number of outstanding omissions in other protocols to meet this demand [24].

DDD is envisioned as a low overhead, highly performant protocol to efficiently move metadata about dataset locations in a distributed manner. DDD is not intended to be an extension of DNS, but instead to be a self contained, non-interdependent protocol taking advantage of multiple concepts in lower Layer existing protocols¹ and apply them to a Layer 7 protocol [35]. DDD provides a method for

content creators to disseminate an arbitrary structure, curated list of Uniform Resource Identifiers (URIs) cross-referenced to a list of DNS or IP entries to source a dataset. This allows, as with DNS to IP translation, for an end user to not need to understand the necessary overhead or end location that is required to provide a requested dataset. As an example, a piece of information could be stored in a date format such as “example.com/src-database/auto-dump/optional/test-20210203.txt” but the system operator can abstract this URL to be a URI of “example.com:/2021/02/03/test” instead. This process is conducted independently of the storage infrastructure. This information, if stored at multiple URLs, would then be loaded by the system operator into the DDD server and any client could make a request to determine which location is closest for a dataset, based on the servers location. However, individual manual entry would not scale well nor would a single server be accurate for most users requests. Therefore, to overcome this issue and remove the burden of adding a DDD entry manually to a server each time it changes, DDD has a functionality to create a distributed, decentralized communication path that can be subscribed to for updates. These connections, which are the choice of each DDD

✉ Joseph White-Swift
joey@utdallas.edu

¹ The University of Texas at Dallas, 800 Campbell Dr., Richardson, TX 75080, USA

¹ Many of the protocols that have influenced the creation of DDD will directly reference the Internet Engineering Task Force (IETF) Request for Comments (RFC) documents from where the protocols have originated.

server operator, can be initiated directly to form a direct trust relationship or by querying other DDD servers to form a recommender trust relationship [1]. While these communications create additional bandwidth usage on a network, the end goal of providing lower aggregate bandwidth usage and faster downloads of datasets for end users can be achieved, as detailed in this paper.

2 Background information

The movement and retrieval of datasets today results in a number of inefficiencies. This is likely due to the fact that the cost of storage has rapidly decreased over time and technologies to account for the abundance of inexpensive storage are only beginning to catch up. The price, for example, of a 3.75 MB hard drive from IBM in 1956 was \$34,500 [22]. However, by comparison an 8TB drive today can be procured for \$155. In the course of 65 years, the price per MB in United States (U.S.) Dollars has dropped from an inflation adjusted \$95,061.15/MB to \$0.0000194/MB, or a drop by a factor of just over 4.9×10^9 . This has resulted in intentional policies of storing as much data as possible, and in multiple locations for redundancy and ease of access [42]. While this data is often understood to be cached or duplicate data, there is a real push to save certain types of data (such as transactions and smart sensor data) to develop new insights based on the large sample sizes. The approaches in Big Data for developing these insights range from data mining, to time based tuning, to certainty qualification for previously developed business insights. All of this data needs to be stored and easily accessible to compute workloads in diverse locations.

However, the lack of a source of truth for this data leads to inefficiencies. This is by virtue of the fact that there does not exist a uniform interface for providing metadata on datasets to end users. Without such an interface, users do not have a readily available source of truth on link costs to query when determining the best physical site to download data or otherwise interact with a dataset. Link costs are a concept that originated in graph theory and are a key factor in finding solutions to the shortest path between servers [26]. This mathematical approach involves the quantification of weights on the edges of a graph. Non-specifically, the weights on the graph are associated with any Quantity of Interest (QOI) in a graph, which could include the length of the edge, the time it takes to travel the edge, or more exotic factors like how one edge compares to another during its traversal (such as having two streets, one brand new and one with potholes). In the context of this paper, the weights on the edges refers to either the latency impact or the bandwidth usage, but in principle this could

be any QOI such as hop count or bandwidth availability. The latency cost is what is required to get between two destinations (broadly) or a summation of individual costs between all of the destinations on a path (specifically). The bandwidth cost is the impact to the available bandwidth when overhead is introduced by DDD that becomes detrimental to the links by consuming a portion of the overall link/edge bandwidth.

It is almost impossible, therefore, for a dataset requester to intuit the costs of a link, as the conditions that combine to create such a metric are seldom exposed to users. As a result, due to the lack of a perceptive filter to the factors of these costs, users inevitably will choose locations to source data that are not the most network efficient (as shown for a few practical cases presented in Sect. 6). Take for example the CLOUD Experiment at CERN [44]. In the topology presented for their intended work flow, the data is collected and made available at CERN and mirrored to the University of Lisbon. This is done for redundancy purposes, but a byproduct effect of providing multiple sites is that there is now a choice for which location to query. CERN collaborators are then expected to download the data from one of these sources. However, depending on an end users location in Europe, would not the best location be the closest one? Or if the data was being utilized in the U.S., the assumption by many users would be to go to the closest physical source which would be the University of Lisbon. Both of these answers could be wrong. Depending on how the users' internet is routed within Europe, and in the case of the U.S., which undersea cable is traversed can opaquely effect which link² would have the lowest cost. For U.S. users, if more than a handful download the data, having a site that acted as a mirror in the U.S. (that is independently in need of the data, too) could both speed up the downloads and lower the aggregate bandwidth required for the undersea cables. Yet, to date there is no use case agnostic internet protocol that is capable of providing such intelligence to end users transparently while simultaneously being capable of receiving updated information to ensure that the protocol can remain the source of truth.

3 Related works

While nothing exists that has solved the problem outright, the problem has existed for a number of years and various ideas have come about to try and solve it.

RFC 1794 details work that was conducted by the IETF DNS working group to provide the ability to parse DNS zone updates and push them to secondary DNS

² The term “link” is used in this paper to refer to the connection that is used by the user to reach the site of a dataset.

servers [10]. This work included modifying BIND servers to allow for transferring volatile or otherwise fast changing information to other servers. The results were positive but not overwhelming. While the approach worked, it was computationally expensive and was not optimized well to handle an increased number of zones. Additionally, issues arose with the Time To Live (TTL) values and concerns that as these reflected fixed time instead of number of hops, the time taken to propagate could exceed the time to live. The solution was to exclude the transfer of this volatile data instead.

Work was conducted at the University of Tennessee, Knoxville in conjunction with Internet2 to develop a concept of “internet channels” for an Internet2 Distributed Storage Infrastructure (I2-DSI) [6]. The channels would be run by Internet2 and consist of replication, distribution, and end user resolution. The intent was for this system to be funded by both government grants and those posting content. The system that was proposed was ambitious, but tried to do everything at once. Each of the pieces of I2-DSI that were detailed were required for the system to work, where no major subsystem could easily stand on its own without loss of features or requiring modification. The system that was proposed did have a simple interface, but failed to provide more concrete definitions that could be applied to alternative use cases.

Internet Backplane Protocol (IBP) was proposed as a way to provide distributed storage including both storage and network devices [5]. This solution takes on two parts, that of the protocol itself and the broader topic of Logistical Networking. IBP aims to control the storage and provide a facility to cache data at different locations, preferably close to the source or destination. While the focus of IBP is on instream data, or buffering data, DDD is focused on pointing to a dataset that resides on end points and is less transient. IBP is limited to IP address for URI discovery which could cause conflicting results on internal networks, where DDD uses a node parameter agnostic naming scheme. Finally, no consistency or security for communications is described in IBP in contrast to DDD hashing each packet. The Logistical Networking revolves around building a logistical session layer (or similar) between the IBP servers to allow for an abstracted flow of data [7]. While this is analogous to DDD having server to server communications to transmit dataset information, they are fundamentally different as DDD does not aim to control the flow of data, only provides pointers to it. Additionally, DDD provides a parameter to the user to directly determine the closest endpoint, where IBP can cache data but must make assumptions about the network layout in order to provide access to this data. Combining DDD with IBP could provide a solution that has knowledge

of the network characteristics and then uses this near real time data to make choices about placement.

A peer-to-peer concept for DNS was presented out of MIT named Distributed DNS (DDNS) [13]. The implementation relies on CHORD based distributed hash tables to provide distributed functionality. This work provided some key insight, chief of which is that this approach requires a much greater number of RPC connections than that of traditional DNS. This overhead led to noticeable decreases in performance in DDNS as compared to traditional DNS. Without any gains elsewhere (such as aggregate bandwidth savings), the costs are not easily justified. One positive insight (parroted in DDD) is the lower amount of work that is required for the server operator to handle large numbers of hosts. Another shortcoming of this approach is the load that is placed on the client that makes this implementation not compatible with existing DNS implementations.

Giggle was presented as a service to catalog datasets and make them available to the scientific community [11]. Giggle aims to be a storage aware system with multiple layers of indexing, split into two parts. These are local replica stores (LRCs) and replica location indexes (RLIs) where the data on a site is stored on the LRCs and users query some level of RLIs that then query the LRCs. DDD, in contrast, flattens this approach into a single server that can be configured to serve both or a single purpose, with the servers being able to query other pre-configured trusted servers for information not known to the server. By flattening the architecture, the source of truth for small datasets can be shared more efficiently due to fewer nodes required to transfer the same amount of information. DDD can also flexibly allow for splitting these two tasks in larger deployments and due to the directed update functionality in DDD (Sect. 4.2), servers can be configured in a high availability mode. This solution can be extended to also push to a shared central server for lower bandwidth usage during requests (such as a University system maintaining a source of truth updated by its member schools) by acting as a consolidated source of truth so that each location (in this case member schools), would not have to be queried individually. The Giggle paper states “An RLI must support authentication, integrity and confidentiality” but does not state how this is achieved in their environment. In contrast, DDD sets out as part of the protocol the necessary key exchange to establish trust (Sect. 4.3) and provides for each message to be hashed to prove authenticity (Sect. 4.3). Finally, DDD provides both a functionality to confirm a returned source is accessible and the ability to test to determine the most advantageous location (based on a parameter such as latency) in order to return not only a site that can respond, but also the closest site to the user.

NetSolve is a grid processing system proposed out of the University of Tennessee Knoxville in the mid 2000s [36]. The concept, much like DDD, relies on a agent broker to determine a list of suitable resources. However, these resources differ in a key way where DDD focuses on locations of datasets, NetSolve focuses heavily on computational resources and Grid computing. Additionally the net result of DDD and NetSolve are fundamentally different. NetSolve is intended to be used with scientific software to offload various computational functions to systems in diverse locations. DDD, by contrast, focuses on providing pointers to dataset locations without understanding of underlying infrastructure.

More recently, the concept of Named Data Networks (NDN) has been proposed to solve a number of issues with the IP structure [46]. However, while some sources claim that NDN is a replacement to the IP layer [45], when looking at the original specification, it notes that “the core IP routing protocols, BGP, IS-IS, and OSPF can be used as-is.” The other core tenant of NDN is that work is done by the router or a similar device to create the caching that is in place for the data. However, this has one, possibly two, shortcomings when it comes to datasets. The concrete issue is that the data objects are placed on the devices as needed and much work is taken to develop algorithms’ that account for how long to keep this data and how to move this data should a device move (such as a mobile device or an autonomous vehicle). This approach requires investment in storage at the router site, which potentially could be expensive to install (in cases where the router enclosure is in the field) and to maintain (as drives or other components fail). This also could require large bandwidth costs as moving datasets that are requested enough to meet the algorithms parameters would not likely continue to have enough requests, therefore making the approach costly for datasets that are large or not used often. An additional possible issue (depending on definition) for datasets could arise if NDN was positioned to replace IP as some sources suggest, as the cost of doing so would be enormous, and potentially so much so that the industry would not take advantage of the new technology.

An addition to DNS was added in RFC 7553 to include a concept of DNS URI translation [15]. The goal of this extension is to provide increased flexibility to support generic, predefined URIs. However, this support is only limited to protocol type. As an example, a URI entry can be added to handle any ftp request, but is limited to existing types of URIs. While there is support for multiple entries and weighting, the weighting does not take into account geographic location.

Rucio, written and maintained out of CERN, has a number of diverse features including some that overlap with DDD [3]. However, as shown in Table 1, this overlap

is not complete and the two systems present a few major differences. The first is that Rucio relies on a centralized infrastructure to define access and control content. DDD, in contrast, has no requirement for access control and instead is defined as a protocol that does not require the additional overhead that Rucio needs for all of its various services. The expectation, as with other interoperable protocols, would be that other layers (such as a VPN) would be used to control internal vs. external access. Rucio centralization also relies on understanding the underlying storage and providing authentication, which leads to bloat when reused. By comparison, DDD focuses on providing location data as compactly as possible in a protocol format so others could write their own implementation if needed. While in some ways the atomic, centralized nature means that URIs in Rucio are valuable on the one hand, on the other they do not allow different Rucio domains to cross (the result could be similar to crossing two internal networks that share IP space). DDD allows for this by accepting the convention that each user will only provide URIs within a FQDN they own. DDD additionally is only used to reference data that already exists, so the assumption is made that the storage is available for the pointers already loaded into it. Rucio does take telemetry data, but this is processed after the fact to determine the best place to store data based on usage. DDD, in contrast, takes regular measurements of the actual network connection and provides the client with the best link based on a predefined quantity of interest.

A brief review of similarities and differences are provided in Table 1, with the main contributions of this work being: (i) Providing a lightweight client interface, (ii) The cost of overhead justified by other savings, (iii) DDD works on top of existing Layer 1 to Layer 4 stack, (iv) DDD is lightweight and storage source agnostic, (v) Supports the union of disjoint sets of DDD services, and (vi) DDD has a concept of near real-time network driven locality that provides a source of truth to efficiently download datasets. The impact of this approach to these existing limitations, as shown in this paper, is to create a protocol that serves as a source of truth for each dataset in a self contained way. This protocol works now, on existing hardware, and is lightweight enough to be included into future software transparently to the end user.

4 System description

The DDD Protocol can be divided into two fundamental segments, the Client/Server and the Server/Server interactions. These interactions follow two different models, with the Client/Server following DNS and the Server/Server following Route Peering. In Fig. 1, a high level description of DDD is presented. The system as described below is

Table 1 Comparison of major features of DDD vs. related works

Feature	DDD	DDNS	DNS URI	Giggle	I2-DSI	IBP	NDN	NetSolve	RFC 1794	Rucio
Authentication										x
Check service	x							x		x
Data/storage aware				x	x	x	x	x		x
Decentralized	x				x	x	x	x		
Distributed store	x	x		x	x	x	x	x	x	x
Membership required										x
Protocol	x	x	x			x	x		x	
Provides closest	x									x
Reusable URIs	x		x	x						
Server side crypto	x				x		x	x		x
Supports URIs	x		x	x		x				x
Union of disjoint sets	x	x	x	x			x	x	x

Fig. 1 Overview of DDD server interactions

generic enough to support any language from FORTRAN 77 (since everything is intentionally in all caps) to Python. Each major section describes in detail how the protocol operates with a model use case described at the end.

To begin with, Client/Server interaction is initiated when the client (c), makes a request for information to the local server (a). At this point the server follows the state machine described in Fig. 2. Once the server finds the most advantageous link, this information is returned to the client (c). This then requires a second request to the data source (d), which is not handled by the protocol (but could be by an integrated client). As shown, more in depth detail on the Client/Server interaction is presented in Sect. 4.1.

For a Server/Server interaction, any DDD server can act as both a source of data or a sink for data, as described in Sect. 4.2. As shown internally on the remote server (b), if data is acquired it will be filtered to determine what is of interest to the server, and can potentially be rebroadcast to other servers that listen to it. Anything that is kept is added to the URI table internally, where the lowest cost link can

then be determined. The Server/Server communications are presented in two parts, with the internal peering infrastructure being described in Sect. 4.2 and the Peering Requests (in the form of packets) being detailed in Sect. 4.3.

4.1 Client to server

The connection between Client and Server mirrors that of a traditional DNS connection.

Figure 2 displays a diagram of the request lookup process. For the client server half of the model, the server sits idle waiting for a request. Once the request is received the DDD server proceeds to determine which dataset location provides the lowest cost link, based upon previously recognized and verified sources, which include

- **Static** Any information that has been loaded directly into the server,

Fig. 2 DDD client/server interactions

- **Cached** Any information that has been discovered by the server when other options are not available, and
- **Peered** Any information that has been shared directly or indirectly with the server.

Each of these data sources are routinely updated to ensure that the DDD server is providing information using current link costs. These updates are undertaken separately but complementary to the client server model. Once enumerated by type, the server sorts each known dataset location to find the lowest cost link. The lowest cost link is then tested to ensure it is still up, and this link is then provided back to the client. If the link is not available, then the server goes link by link down the sorted list until an available one can be provided to the client. The default action is to return the IP address of the available data along with the remaining sub-directories in the URL that are required to reach the dataset. This is to ensure that if a DNS address has multiple IP addresses, each of those IP addresses can individually be tested and the lowest cost link determined. Due to this, it is important that a “local” DDD server be utilized, as link cost information is related to the server and does not take into account any conditions of the client. The closer the local DDD server is to the client, the more accurate the returned value will be. Therefore, ideally there would be a DDD server either running on the client machine that could be used to make these local lookup requests, or in the case of an institution, one DDD server could be run for an entire campus.

4.2 Server to server peering functions

The server to server peering follows an approach based on route peering. As with RIPv2 or OSPF, when a new record is added or updated, that information is distributed [21, 28]. To avoid the problem of constant bandwidth usage that is generated by RIP, the “send only with

new data” approach of OSPF is borrowed in the DDD protocol implementation.

4.2.1 Source sink topology

The peering in DDD follows the concept of peer sources and peer sinks. While they both share the peering port, each has a unique function. The sink is responsible for listening to data from other DDD servers and working to process this data once it arrives. This processing includes handling the different requests in Table 2 as well as validating that the signature that is received by the server is valid. These requests are then filtered through the peers of interest (see paragraph below) to determine if the server chooses to add this URI to the peer entries. If the server does not, it either returns to listening or rebroadcasts them based on operator preference. If the rebroadcast is selected, the record is handed off to the source for further action.

The source then takes new information added to the server and pushes that out to sites that are interested. This is done for all of the servers that indicate YES in their Start

Table 2 Peer-to-peer message types

Type	Function	Brief description
SOA	INIT	Trust/Key Exchange Request
SOA	UPDATE	Push Updated Crypto Keys
SOA	ACCEPT	Denotes Successful Key Entry
HELLO	N/A	Ask for Updates Since Last Boot
A	ADD	Add a IPv4 Record—Peer Table
A	MOD	Modify IPv4 Record—Peer Table
A	DEL	Delete IPv4 Record—Peer Table
AAAA	ADD	Add a IPv6 Record—Peer Table
AAAA	MOD	Modify IPv6 Record—Peer Table
AAAA	DEL	Delete IPv6 Record—Peer Table

of Authority (SOA) request as described in Sect. 4.3. Additionally, any information that is determined to be real by the sink and passed over to the source is pushed out to the various SOA records as well. The source and sink can operate independently of each other. This is to allow for a site that only ingests data (sink only) as well as a site that only provides data for the associated URI (source only).

4.2.2 Peers of interest

Peers of Interest (or POI) allow a server to filter the information that is passing through. The URI's for the POI's are the choice of the server operator and account for which dataset locations are stored. In this way, operators are allowed to determine the records that are captured, thereby preventing a situation where an operator that is interested in five resources does not have to account for hundreds. However, with rebroadcasting (see Paragraph below), these choices do not cause a detriment to other DDD servers on the network by filtering information that is of interest to those servers.

4.2.3 Rebroadcasting

Rebroadcasting allows information, even if not of interest to the site, to be passed around the network in a responsible way. First, the TX_IP (as used in Sect. 4.3), is provided in three types, as shown below:

1. **Local global** A local global record is one with a NULL IP address in the TX_IP and a TTL greater than one. If viewed from the server on which it was created, the record is just that, local. On an receiving server (via the sink), however, it is viewed as a local global record in that this is the first hop that is receiving the information. This NULL IP is to confirm a direct trust, and it is then the responsibility of the server to fill in the IP address of record before it becomes peered again, in which case it becomes a Remote Global record [1].
2. **Remote global** A remote global value is one that will have a valid IP address in the TX_IP and a TTL value greater than one. This indicates that the information is not originating from the server transmitting it, but instead from the server listed in the TX_IP. This is an example of recommender trust that exists between the originating server and the receiving server [1].
3. **Directed** A directed record is one that will only go to another DDD server directly, and is denoted with a TX_IP value of "YOU" and a TTL of zero (TTL = 0 is reserved for this purpose). This is intended for internal consistency between servers operated by the same operator or to be used to send information to a

central peering server for lower overall bandwidth usage.

Additionally, a TTL is set on each server to ensure that broadcast storms are minimized. Each server that rebroadcasts a record is responsible for decrementing the TTL value by one before sending the information to the SOA's that have requested updates. If a sink receives an update that has a TTL of one, then that update is considered exhausted and should not be placed in the Rebroadcast queue with TTL zero (which, as noted above, is a reserved value).

4.3 Server to server peering requests

Table 2 lists the different request types for the server to server peering, the sub-functions that exist in each, and a brief description for each sub-function.

The majority of the messages are signed using the Blake2 algorithm, an algorithm that is speed equivalent to MD5 while providing more secure connections than SHA1 [14]. Each of these peer to peer functions is explained in greater detail in the following sections along with other key operations.

4.3.1 SOA request

Borrowing terminology from DNS, the first type of DDD Request is the Start of Authority (SOA) request [25]. The SOA request in DDD is used to establish dynamic, distributed trusts between DDD servers. Each DDD server will generate a public/private key-pair to use with all interactions between servers for signing. This ability to create trust dynamically allows arbitrary federations of servers to form topologies such as mesh, ring, star, tree, and hybrids of each. In doing so, the first communication between two servers is through a SOA request since there is a lack of central authority in the DDD environment (as opposed to root servers in DNS). These interactions are stored as SOA records on the server, and the process is shown in Fig. 3.

The first interaction between two DDD servers is an INIT request. This request, which is not signed, is sent with the form:

SOA INIT <LOCAL_PUBLIC_KEY> <Updates>

The receiving server then processes and stores the key it received for later use. The updates are either set to YES or NO, determining if the "local" server is interested in receiving updates from this server as records change. In order to tell the "local" server that the key has been accepted, the "remote" server will reply back with an ACCEPT packet in the form of:

Fig. 3 DDD SOA flowchart

SOA ACCEPT <REMOTE_PUBLIC_KEY >

The originating server is told by this one transmission two things. The first is that the remote server accepted the key it was sent and the second is the remote server's key. The "local" DDD server then stores that key to complete the key exchange and trust is formed. All future transmissions between the servers will be signed, greatly reducing the chance of man-in-the-middle attacks [18].

Once trust is established, the system will then sign every message going forward. Understanding there will be cases when this cryptographic information should be updated, there is an UPDATE command to facilitate any required changes. The UPDATE command takes the form of:

(Signature) SOA UPDATE <NEW_PUBLIC_KEY >

The server that receives this command will first check to make sure that the signature matches, and if it does, update the public key for the peer. This ensures that once a trust relationship is created, every single change or update can be confirmed between both parties. To confirm that this new key has been loaded, an SOA ACCEPT is sent back.

4.3.2 HELLO request

Borrowing terminology from OSPF, the next type of DDD Request is the HELLO record [28]. As in the OSPF parlance, the HELLO request is used to query another "router" (in this case a DDD server where there is an existing SOA record) to determine what has happened since a previous internal server timestamp. The "remote" server then begins to send over data that matches this criteria. Due to the previous creation of trust, unlike common OSPF connections that require some sort of authentication token known by a network admin, the signed messages provide this trust [12]. This process is shown in Fig. 4. There is no requirement for how often these exchanges need to be run, since if a system is online there are other functions in DDD

Fig. 4 DDD Hello flowchart

(A and AAAA requests' described later in this section) to receive updates that use less bandwidth.

When a HELLO request is first sent out, the server initiates the connection in the following form:

(Signature) HELLO <TIME >

The signature is based on previous trust. The time that is provided to the "remote" server is in the form of epoch time [31]. Epoch time has been around since the creation of UNIX so it is supported by many modern systems, and epoch time is time zone independent [31, 34]. This time is determined by the server and is a request for every local record add or change that has happened on the "remote" server since the previously requested time.

If there is an update that has happened, the server replies back with the record information. Below is the format:

(Signature) <URI > <URL > <Type >

The URI in the message is the URI that the server owns. Since no central authority exists as in DNS to confirm ownership, the responsibility of ensuring this is correct falls on the users. However, because peers can pass on records between servers (see A and AAAA in Sect. 4.3), as long as the source that is peered with is trustworthy, the end user does not need to enter into SOA trust with each server where data is requested. Much like sources of poisoned BGP peers or non-standard DNS records, false DDD record providers are likely to be quickly realized and have little effect in the long term. [33, 37]. URL is the actual location (DNS record) for the dataset. This is provided as a human readable DNS record for ease as well as to support DNS load balancing or IP address changes thereby not requiring a new entry to be stored for the DDD server. The Type is the kind of link that is provided, either static or stream. A static dataset would be a pointer to a file that is not likely to change, whereas a stream could be of any data or other media that is changing.

Once the data is sent to the requesting or “local” server, that data is processed. When the server is ready for the next piece of information, it responds with a:

(Signature) NEXT

This NEXT data process continues until all of the matching records have been sent to the requesting server. Either once this back and forth to exchange records has finished, or if there are no records that have been updated since the requested “EPOCH” time, then the DONE is issued by the remote server taking the form of:

(Signature) DONE

This method of inquiry is designed to aid a server that has been offline for more than a few minutes, but at the cost of a large amount of bandwidth. Therefore, to improve on this (and to eliminate some of the bandwidth issues that can occur with a large number of HELLO requests), the A and AAAA request are used [27].

4.3.3 A request

Borrowing terminology from DNS, the A request in a DDD Request is like that of an IPv4 Address (A) record [25]. The A record stores information for URLs that are based on IPv4. There are three types of A record requests, each falling into the following general form:

(Signature) A <FUNCTION> <URL> <FIELD>
<VALUE> <TX_{IP}> <TTL>

This general form is then used to transmit the three types of operator: ADD, MOD, and DEL. The URL is included in each and is generally considered a unique quality of a record. Each server’s sink waits listening for updates, and if one is received from a trusted source, it is then processed, as shown in Fig. 5. Each message is sent via TCP, and since each record is meant to stand on its own, no NEXT or DONE statement is required to be returned back.

Fig. 5 DDD A ADD, MOD, DEL flowchart

A records are created with the ADD function. The following is the format of an ADD, as it fits into the general form listed above:

(Signature) A ADD <URL> <Type> <URI>
<TX_{IP}> <TTL>

The field in this case is reserved for the type of connection, and the value is the URI associated with the URL. Depending on the rebroadcasting type, the TX_{IP} and TTL are set as described in Sect. 4.2.

A records are modified with the MOD function. The following is the format of a MOD, as it fits into the general form listed above:

(Signature) A MOD <URL> <FIELD> <VALUE>
<TX_{IP}> <TTL>

The field in this case is reserved for the field of record that needs to be updated (such as a URI, Type, etc.). The value is the replacement value that is to now be associated with the URL. Depending on the rebroadcasting type, the TX_{IP} and TTL are set as described in Sect. 4.2.

A records are deleted with the DEL function. The following is the format of a DEL, as it fits into the general form listed above:

(Signature) A DEL <URL> NULL NULL <TX_{IP}> <TTL>

The field and value for a deletion are set to NULL as they allow the DEL to fit into the standard since the URL to be deleted requires no additional attributes. Depending on the rebroadcasting type, the TX_{IP} and TTL are set as described in Sect. 4.2.

4.3.4 AAAA request

Borrowing terminology from DNS, the AAAA request in DDD Request is like that of an IPv6 Address (AAAA) record [39]. The structure, as alluded to in Table 2, is duplicate to that of the A request except replacing that with AAAA. Therefore, as an example, a DEL request would be formed by the following:

(Signature) AAAA DEL <URL> NULL NULL <TX_{IP}> <TTL>

Due to this function, there is no need to support different binaries or functions based on IPv6 versus IPv4, as some implementations have been forced to do (such as trace vs. trace6, ping vs ping6, or at a minimum flags for the different versions) either on the client or server side [30, 40].

4.4 Use case

A simple use case that explains how the DDD protocol could be deployed would be that of organic adoption and the resulting parallel benefits. One could easily conceive that multiple schools in a system, for example the UT System, would want to exchange information between each other for common research goals. A similar system could be implemented by the Texas A&M Agriculture Extension Service to ensure that each location is pulling or pushing datasets efficiently. At some point, users would start to have needs for data that went outside of these university systems, or even individual schools, that could be coordinated by a regional network operator like LEARN³ that would benefit from the decrease in bandwidth utilization directly. While in the beginning it might be a small savings, as sets continue to form and interlink, those savings would be magnified. Additionally, such a regional operator could provide a centralized location to accept directed peers to allow for further bandwidth reduction as there would be a regional clearing house for this information. Instead of having to go multiple places to look for data, this could be centralized and the latencies determined against a users local DDD site once the data pointers are pulled.

However, like any scalable protocol, the larger the adoption the more beneficial the solution becomes. So with time and success it is reasonable to assume that other locations could do the same thing, with regional operators acting as data brokers within an area and local servers only needing to point at them (LEARN, CENIC⁴, etc.) in order to have a full mapping instead of needing to reach out to all the schools in Texas or California. As long as these regional clearing houses were available on the public internet, then anyone could benefit from the information aggregation (or if only on Internet2, then at least academic and research institutions would benefit). At the same time, in tandem, this protocol could be utilized internally within subscription based services and private organizations without the intent to join the larger public DDD network. DDD is a protocol, so this would be possible and could be included in products using the native code of a product.

This could be further optimized by having multi-regional or national DDD servers that could serve as a clearinghouse for the regional operators, run by an entity such as Internet2. The advantage of this evolution would be that there would be minimal hops to the information, but if any single system went down, there would be paths with lower weight that could be utilized to continue to have a resilient network. Over time this could even grow further to encompass international operators. While the down side of

this environment is that there is no centralized source of truth, the individual server operators can choose the locations that are peered, therefore minimizing the impact of erroneous or malicious information. By having back up paths to information, the fear of a single point of failure at a server causing a national outage would be tempered, as adoption of this protocol would happen organically and there would be lower priority redundant paths.

This example shows a couple of the key strengths of DDD. First, as it supports later unions of disjoint sets, growth and adoption can happen organically without the need to have a governing body or centralized authority. Second, since it is a protocol, it will allow for usage within multiple organizations and can do diverse tasks based on the software chosen instead of needing to adopt a whole infrastructure that someone has cobbled together for this purpose. Third and finally, the more insight to datasets that is achieved by the DDD network growing, the more savings that can be provided in bandwidth utilization. This is especially important for network operators that maintain undersea or backhaul cables, as the ability to provide pointers to local or regional copies of data can mean real savings as opposed to traversing national or intercontinental backhauls to obtain a dataset.

5 Preliminary implementation

For a balance of features (such as sockets, pthreads, and MariaDB integration) and efficiency (both space and time-to-answer), C was chosen for the preliminary implementation of the DDD server protocol. A very basic Client written in C, named dddig as a nod to the DNS dig, is used to test results of the various server functions [20]. By using C, the preliminary implementation can demonstrate how compact DDD can be and how easy it is to utilize within other software. Compared to other similar solutions with available source code, such as Rucio, DDD is 26 times smaller (in terms of the executable footprint) [3]. The total server implementation, as described below, only contains 1660 lines of code and yields an executable file that is 279 kilobytes in size. Due to so many security issues originating on the network, using C for the standard libraries, small line count, and protocol compactness makes security and audit verification straightforward. To this end, the basic client is written to be the most minimal size possible to interface with the server and only requires 60 lines of code and is 17 kilobytes in size.

Additionally, with its symbiotic relationship to UNIX and its derivatives, using a C implementation means that little effort is required to execute the program on most operating systems today and while there will be some difference in the operating systems, the results presented in

³ Lonestar Educational And Research Network

⁴ Corporation for Education Network Initiatives in California

Sect. 6 should be very similar across UNIX-like operating systems [34]. To aid in portability, the code is created with the UNIX philosophy in mind, drawing heavily on the portable UNIX Sockets interface and only including the bare minimum of required executable options to help cut down on bloat while still allowing for the preliminary implementation to prove how flexible a DDD server implementation can be [29, 38]. While it is the choice of the authors to use C, the protocol presented is code agnostic and can be written in any language that can meet the specification that is described in Sect. 4.

Both executable files are built using various minor versions of GCC Version 8 without optimization. The operating system used for development is Debian 10. Testing was conducted on both Debian 10 and Ubuntu 20.04 systems. This development and testing include operating in containers that have 100MB RAM limits and the system, including all associated components, experiences no problem meeting this self imposed limitation. In the Ubuntu case, no change is required to the code for it to run in that environment, and even the libraries have the same names. The Linux distributions used are based on personal preference. There is nothing inherently preventing this DDD Protocol implementation from running on other distributions.

5.1 Major differences or omissions

The preliminary implementation focuses on the core functions of the DDD Protocol as outlined in Sect. 4. While the implementation accounts for most functions, some have been omitted in this first release. The most major omission is the support for IPv6 addresses. While the code and data structures required to differentiate between IPv4 and IPv6 are non-trivial, they do not impact the ability to test the core functionality presented in Sect. 6. Partial support exists for the POI filter, but to simplify testing all records are currently allowed. Also not included, is the requirement that a server can go to a “server of last resort” to find cached information. This relies heavily on the POI filter to ensure that there is a concept of where to search for information that a server does not have knowledge of previously. To minimize testing time and to increase testing options (such as to account for when websites change locations or do not allow ICMP), testing to confirm that a requested entry is up is not implemented. In most testing cases, the records are added and the server immediately queried, so efforts are focused on getting the right answer back since functionally the server just tested that the server of interest is up.

While the standard has a functionality to dampen broadcast storms (through the TTL), additional measures are taken to decrease the likelihood of broadcast storms.

Broadcast storms, a detrimental network condition, can be caused by a device that is flooding the network with broadcasts, a broadcast request that is answered in unison, a large attempt to forward broadcasts, or some combination of all of these factors [23]. Since there is a requirement to keep track of the transmitting IP address for a record (TX_IP), the information is used to prevent peering information from ever returning to the original sender. To expand on this concept, this implementation also captures the IP address from where the record was presented (the RX_IP) and applies this as well. The results of this choice are further discussed in Sect. 6.2.

5.2 Implementation choices

While the standard suggests that data such as the SOA keys be stored (Sect. 4.3), there is not a defined storage routine. For all of the storage, MariaDB is chosen as the retention backend. This is due both to the ease of adding MariaDB to a system and to well-defined C API library that exists for producing these functions [4].

The following list describes the different table names and the uses for each:

- **static_entry** Any information on records directly entered into the DDD server,
- **peer_entry** Any information on records peered into the DDD server,
- **soa** Records on trusted peers and which direction the trust is built,
- **source_spool** Spool for outgoing information,
- **crypto** Storage for the local private and public key, and
- **times** Epoch time storage table for persistent time stamps.

Monocypher is chosen as it implements Blake2 hashing algorithms, requires no dependencies, and is written in C [41]. Adding this code only requires downloading and including the header file in the appropriate section of code. Due to the generic requirements of the standard, the initial implementation does not take advantage of the built in cryptographic key exchange process presented as part of Monocypher.

While the specific Quantity of Interest is not stated in the standard, the initial implementation includes the functionality to ping the available hosts. The assumption is made that the datasets discussed are publicly available research data and therefore are accessible to a ping request. For the initial implementation, liboping is chosen as its functions are very flexible and available for C [17]. As with the ease of using MariaDB, the choice to use liboping is made easier as there is support for a wide range of operating systems.

In order to allow for the different functions of the server to operate independently, pthreads (short for POSIX Threads) are chosen [32]. The pthreads are also chosen for the ability to split the function of the code into different loops for timing and to allow both ports to be bound at the same time. While the possibility exists to use pthreads for additional parallelism, in the initial implementation the focus is to provide a way to test the core functions of the DDD protocol. If all of the functions are enabled in the daemon, then five threads are created. For the Client/Server connection, a thread is used to handle all interactions. For the Server/Server connection, a thread is first created that then spawns two additional threads, one for the sink and one for the source. The final thread is used in the main daemon to handle “housekeeping” tasks, such as parsing SOA INITs and spawning updates for out of date records.

In addition to the daemonizing function, the server executable contains other functions to interact with the data structures. There is an “init” function for initializing all of the database structures so the system operator does not need to. There is also a concept of a separate set of functions to input data into the server, such as adding or deleting a record. These functions are designed to be invoked with the same executable code as the already running daemon (for compactness, only one executable needs to be built), and the running daemon has the intelligence to not be affected by these “live changes.” The executable code is designed to be self contained and easily placed into high availability. Only the database changes need to be tracked, and the configuration file needs to match between both the primary and fallback location. The configuration file, while basic, allows server options to be set as well as the ports to be changed during testing. This approach is necessary with a compiled language like C, so that simple tweaks or large scale testing do not have to rely on changing and recompiling the underlying code.

5.3 Bandwidth usage

While each system should follow the “standard” formatting of DDD, the initial implementation makes some key decisions that affect the bandwidth usage equations. To begin with, the Client to Server (Sect. 5.2) is written using TCP protocol versus the more appropriate UDP. This choice is made for simplicity (since the code for the Server to Server connection could be reused) and to have guaranteed transmission in the environment. The equation for the Client to Server connection is based on each side having a fixed 400 byte long data field, which including overhead is 468 bytes. Future implementations could use

variable packet size to further lower bandwidth, but 400 bytes is chosen to fully encapsulate any data to return to the client. The bandwidth usage due to the interaction as implemented therefore can be generally modeled as follows:

$$BW_{client} = (n_{answers} + 1) * BW_{cc} \quad (1)$$

where $BW_{cc} = 468$ and $n_{answers}$ is the number of returned values. As the result of certain Client to Server functions not being implemented (such as the ability to return all known values), the number of returned values ($n_{answers}$) would equal one. Therefore, the total bandwidth used is simply twice that of the fixed frame, or 936 bytes.

For the Server to Server peering, the packet size is set to a fixed 740 bytes of data, which yields a total size of 808 bytes of data when including overhead. It would again be best for a future implementation to utilize a variable-sized packet to decrease the aggregate bandwidth usage, but 740 bytes supports the longest possible URI, a change to that URI, and the hashing of the message. Therefore, the numbers presented below for the equations should be considered those for the “worst case”, as all of the values that make up these packets can never be larger than 740 bytes.

The following equations represent the bandwidth utilization of each type of peer to peer function:

$$BW_{soa} = 2 * BW_{sc} \quad (2)$$

$$BW_{hello} = ((n_{records} * 2) + 2) * BW_{sc} \quad (3)$$

$$BW_{diff} = n_{soa_peers} * BW_{sc} \quad (4)$$

where $BW_{sc} = 808$, $n_{records}$ is the number of records, and n_{soa_peers} is the number of SOA peers that previously requested updates for this URI. For the SOA INIT bandwidth, BW_{soa} , a key exchange takes place and therefore it only is two times BW_{sc} . This is conducted once to build trust, but is not a recurring bandwidth cost. For the HELLO request bandwidth, BW_{hello} , each record is sent and waits for a “NEXT” command, plus the required initialization request and “DONE”. This means that each HELLO request requires a minimum of two times the BW_{sc} , and grows linearly with the number of records. For the ADD, MOD, and DEL, collectively modeled by BW_{diff} , the number of SOA peers that request updates will drive the number of updates that must be sent out. However, unlike the HELLO request, there is no confirmation packet that is required by the source.

For more information on the parameters of these requests, please see Sect. 4.3.

6 Results

Experimental results are broken up into two sections, with the first (Sect. 6.1) for the Client to Server communications and the second (Sect. 6.2) focusing on the Server to Server communications. Additional results for Sect. 6.1 can be found in Appendix, Table 7.

6.1 Client to server

While the results are, to a degree, determined by the implementation, appropriate procedures have been followed to generate them in a rigorous way. Each InstaGENI site is loaded with an experimental data set containing twelve entries [8]. These entries are for three files stored on the same four Wasabi S3 buckets. The buckets: US-west-1, US-central-1, US-east-1, and EU-central-1 are chosen for geographical dispersion and to include both the United States and Europe. The three sites in the U.S. are located in Hillsboro, Oregon; Plano, Texas; and Ashburn, Virginia, respectively [43]. These three establish a diverse test set to determine the accuracy of the returned latency results, presented further in the paper. The EU-central-1 site is established to demonstrate the increased cost of travel ling over undersea cables, and is utilized to confirm that requests originated from the European Continent would benefit from storage in the EU compared to storage in the U.S. Each Latency is tested using the average of three pings for each IP address returned from a DNS request for the URL loaded into the DDD server (either by direct static input or via a peer). By implementation, these records are updated every 24 hours to account for changes in the environment, although a smaller value could be used for greater accuracy or a larger value could be used for lower bandwidth consumption.

Large scale testing was conducted to validate that the results provided from the DDD servers reflected the lowest cost. 34 GENI sites are used to test against all four test storage points. The three storage sites in the U.S. are designed as triangulation points in conjunction with the extrapolated physical location vis-à-vis GENI Site name (such as Michigan State University would be located in or around Lansing, Michigan). Table 3 reports how many sites are accurately reported as First, Second, and Third closest sites based on geographical assumptions. In determining

the geographic location for our data, the following differences were used about the locations based on information provided by the GENI Operators: i) The University of Texas InstaGENI is a typo for The University of Texas at Dallas, ii) GENI maps show that “metrodatacenter” is located in Ohio, and iii)BBN Technologies is based in Cambridge, Massachusetts.

Additionally, 32 out of 34 sites (or 94.12% accuracy) reported the EU-central-1 site as last, which was the anticipated result. All 34 (or 100%) of the sites returned a U.S. based entry in first and second site. The full latency information can be found in Table 7 in Appendix.

While almost all of the sites follow the geographic breakdown of sites intuitively, three sites do not follow this geographic distribution. To begin with, in Fig. 6a, UCSD is shown as an example where each of the locations have increased latency as the geographic distance increases. The InstaGENI in Hawaii, by contrast, prefers going all the way to US-central-1 as opposed to the expected US-west-1, with the round trip time being over 9 ms faster, as shown in Fig. 6b. While on first glance this looks to be an error, upon deeper inspection via traceroutes the reason is apparent. In the route that is followed from Hawaii to US-central-1 in Texas, the link crosses the undersea cable to California and then gets routed over the private Flexential data network. Interestingly, Internet2 does have very close connections into Dallas (tested between the UT Dallas and Hawaii GENI locations), but this faster connection is not used. For the connection from Hawaii to US-west-1, the traffic takes a different path and undersea cable to Seattle, where it is then routed all the way down to Los Angeles before reaching a commodity network to go back up the West Coast. If this path instead went over the cable of the first connection and got handed off there (as the other connection does), it would likely have the intuitive US-west-1 connection first on the list.

The two test sites that put the EU-central-1 storage target, in Amsterdam, ahead of one of the three U.S. based targets, are BBN, Fig. 6c, and Princeton, Fig. 6d for position three (one and two are still both the expected U.S. based locations) [43]. Initially it was assumed that Princeton’s proximity to New York Undersea Cables might be a factor in this, yet multiple New York test points still had the three US targets first. However, by following the route, it was determined that the Princeton connection drops down to the Internet2 Washington D.C. router before going back to New York to hop the undersea cable operated by Hurricane Electric (HE). This routing choice provides this link with an added benefit since Wasabi peers directly with HE in Amsterdam. In contrast, the traceroute for US-west-1 runs through very few routers on the way to the hand off in LA. While in every other instance the traffic goes directly to a private provider, in this case once the

Table 3 GENI/USA geographic accuracy statistics

Type	Accurate	Tested	Accuracy (%)
First	33	34	97.06
Second	33	34	97.06
Third	32	34	94.12

Fig. 6 Results from 4 example sites where the dark bar represents latency and the light bar represents the distance in miles as the crow flies. Site **a** is UCSD, **b** University of Hawaii, **c** BBN, and **d** Princeton University

hand-off happened the traffic is routed through a node that adds 36 ms to the connection before the hand-off to the private provider. This mystery IP address that is adding latency is in a block owned by Internet2, but searching through the routing table in the core router that handled this yielded that it should be following the default route. BBN, instead of following along Internet2, uses commodity internet. In both cases these connections are furnished by Cogentco. The connection to the Europe storage site goes Boston to London to Amsterdam using a very low hop count. In contrast, the latency cost for the US-west-1 connection has to do with the indirect connection path. First the traffic enters the network in Albany, New York (from the Massachusetts area) and then proceeds to take a long and meandering eight hop path to finally be handed off in Portland, Oregon. Due to both the number of links and inefficient hops, the amount of latency required to get to Seattle is equivalent to getting to Amsterdam, with more than twice the hop count. All three of these outliers prove that while intuition might have pointed to a geographically closer location, by removing human intuition from the dataset discovery and acquisition process, and instead using real network conditions as the source of truth, the

factually best options are presented to end users taking advantage of the DDD protocol.

Table 4 shows the testing that was conducted by colleagues in three different European cities. The objective is to test if the DDD test locations would behave in the opposite direction and choose the EU location before any sites in the U.S. In each case, the latency that is returned as the lowest by the three running servers is for the EU-central-1 storage target in Amsterdam, NL. While this sample size of three sites is low, the 100% accuracy in combination with the 94.12–97.06% accuracy (as shown in Table 3) of the large scale US testing provides clear evidence of the accuracy of this approach.

6.2 Server to server peering

The server to server peering is next tested against the following logical topologies: Mesh, Ring, and Balanced Tree. This testing is conducted on topologies consisting of 3, 4 and 5 nodes, respectively. In each instance, all of the traffic is analyzed based on both the SOA INIT (initial bandwidth to build the topology) and the data cost (how much bandwidth is required to update records). From this data, two metrics are evaluated.

Table 4 European latency data

Location	US-west-1 (ms)	US-central-1 (ms)	US-east-1 (ms)	EU-central-1 (ms)
Heidelberg, DE	145.077	131.849	89.998	10.592
Mainz, DE	246.634	260.463	98.641	28.617
Annemasse, FR	196.723	191.781	157.648	62.793

Table 5 Peer-to-peer convergence—iterations

Size	Mesh	Ring	Tree
3	3	3	1
4	8	4	N/A
5	18	5	2

The first, in Table 5, describes the time to convergence for each for the different topologies in terms of the number of iterations required to notify each node in the network. The time to convergence is calculated by determining the number of connections (for ring and mesh) or number of layers (for tree) and the relationship to the number of iterations needed for the network to become stable and the data loaded into each node. Hence for mesh and ring, if the size is three and the number of iterations is three, then this confirms that the number of iterations required to reach network stability is equal to the number of minimum hops required to traverse the topology. The tree presented is a linear balanced tree; therefore, there is no data for size four and the data is injected at the root of the tree.

The second, in Table 6, describes the bandwidth required to come to a convergence point. This should be thought of as a negative offset to the positive bandwidth savings produced by using a DDD server. This bandwidth does not include the SOA requests required to establish the network, as those are considered a one time cost, where the table details the cost for each record.

The reported bandwidth utilization is greater than the theoretical number based on the equations presented in Sect. 5.3. Due to differences in network conditions (such as TCP re-transmissions), this number cannot be fully modeled and only experimental data provides the true total. The overhead is comparatively low for one sustained connection; however, since each connection is terminated after each data update, the three-way hand shake and “FIN-ACK” account for a large overhead for the system as compared to keeping the TCP channel open for multiple communications [19]. Therefore, the equations can provide a better approximation as the number of records sent over a TCP connection (like a long HELLO request) increases.

For protocol convergence, it should be noted that because the DDD server implementation captures both the original transmitting server, and the server from which a record is received, the behaviors of both the trees and rings

tested are such that no unnecessary packets are sent over the wire. For these two topologies, this choice significantly speeds up the time to convergence as compared to that of the expected time of convergence for a protocol that does not have Layer 2 knowledge for where the information is sourced. In fact, in this testing only one interface is provisioned. This would likely confuse equivalent Layer 2 broadcast storm minimization techniques [23]. For mesh, convergence efficiency is not as well improved by these choices. One positive result, however, is that in each case of three, four, and five servers, the DDD server that originates the peering packet never receives it again, thus lowering the overall storming in the network.

7 Conclusion

The demand for a solution that can lower aggregate bandwidth usage while decreasing download times will continue to grow as the amount of data produced and stored increases. Therefore, the distributed dataset DNS Protocol is presented as a solution to a number of outstanding omissions in other protocols to meet this demand.

The DDD Protocol provides the functionality of DNS for datasets in a distributed, decentralized manner. This allows system operators to determine the best configuration for each running instance, and to easily provide infrastructure that scales to meet demand. The functionality presented in this paper represents the core features that provide geographically aware answers and can easily share dataset metadata. The system relies on establishing trust and continuing to confirm that trust during each transaction. This security approach additionally provides access via separate ports, meaning that DDD can easily be added to any organizations threat model and secured accordingly. The system requirements and overhead for the DDD Protocol are small, meaning that DDD servers can easily be deployed in data centers, on network devices directly, and on home computers and mobile devices with minimal impact on system resources. DDD provides a uniform interface for end users to quickly and efficiently determine the best dataset physical location to use. Simultaneously, the DDD server provides the ability to keep this data up to date with only initial system operator intervention; thereby making DDD an attractive solution for institutions and corporations to adopt. Implementing the DDD Protocol also positively impacts network operators, as less cost is required to upgrade expensive back haul equipment or undersea cables due to reduced aggregate bandwidth usage.

DDD is flexible enough that internal organization’s as well as multiple, co-existent public networks can be built using DDD servers in a variety of topologies. Additional work is necessary to better characterize these peer-to-peer

Table 6 Peer-to-Peer Bandwidth Utilization

Size	Mesh (bytes)	Ring (bytes)	Tree (bytes)
3	5004	2808	2808
4	32,886	3654	N/A
5	220,872	4872	4872

interactions in order to best understand which topologies are most efficient as scale increases. While DDD does incur additional bandwidth usage, with smart implementation and topology choices, these impacts can be minimized. While TCP is currently required in this specification for the DDD Protocol, the overhead incurred by TCP should be evaluated against other protocols that may have lower overhead. The real, tangible savings that can be provided by transparently providing the best link to users, even with this overhead, still has the potential to significantly outweigh the incurred costs of running DDD. While latency via ping was presented in this paper as the link cost determination, further work should be done to compare this to other options such as using HTTP headers for cases where ICMP is blocked or integrating with

additional network aware approaches such as the ALTO Protocol [2, 9, 16].

When users need a use case agnostic internet protocol that is accurate, efficient, and a fault tolerant source of truth, the DDD Protocol provides a novel pathway to lower aggregate network bandwidth usage, decrease download times, and prevent single points of failure when retrieving datasets.

Appendix

Table 7 in this appendix reports the measured latency information for each of the 34 GENI sites with respect to each of the 4 storage sites.

Table 7 US latency data

Location	US-west-1 (ms)	US-central-1 (ms)	US-east-1 (ms)	EU-central-1 (ms)
geni.case.edu	62.027	49.793	11.149	93.437
instageni.cenic.net	23.347	32.843	62.590	138.823
instageni.clemson.edu	81.766	28.648	16.805	95.518
instageni.colorado.edu	29.898	23.638	37.325	110.452
geni.it.cornell.edu	74.948	49.827	14.100	78.981
instageni.nroc.gatech.edu	69.611	21.029	16.474	92.371
instageni.hawaii.edu	93.235	84.234	112.954	192.619
bbn.instageni.net	81.386	50.695	15.385	73.929
instageni.illinois.edu	54.354	32.776	18.629	93.305
instageni.iu.edu	62.248	36.441	22.169	95.840
geni.kettering.edu	66.456	39.639	19.849	99.212
instageni.ku.gpeni.net	57.455	15.778	26.464	102.633
instageni.rnet.missouri.edu	60.490	29.927	32.405	113.632
instageni.maxgigapop.net	79.915	34.236	2.789	82.399
instageni.metrodatacenter.com	57.507	37.510	24.870	99.616
instageni.northwestern.edu	67.677	29.150	15.847	90.540
instageni.nps.edu	16.882	42.288	61.665	138.742
instageni.nysernet.org	73.223	48.172	13.393	76.838
genirack.nyu.edu	69.758	36.211	6.304	74.586
instageni.osu.edu	75.852	38.816	12.913	93.861
instageni.odu.edu	85.334	39.336	6.181	81.549
instageni.cs.princeton.edu	85.916	43.512	9.146	80.863
instageni.rutgers.edu	64.038	41.030	6.671	75.619
instageni.stanford.edu	23.674	40.964	59.915	139.997
geni.uchicago.edu	63.244	28.887	15.695	92.051
instageni.idre.ucla.edu	24.064	33.594	63.452	140.508
instageni.ucsd.edu	25.850	35.628	65.249	142.279
lan.sdn.uky.edu	76.913	45.969	25.127	101.154
instageni.umkc.edu	57.664	11.749	26.479	118.062
instageni.research.umich.edu	51.305	37.084	17.209	96.965
instageni.utc.edu	70.687	24.861	20.112	97.530
instageni.uvm.edu	69.191	47.593	13.453	79.853
instageni.arc.vt.edu	74.055	42.783	9.509	86.854
instageni.wisc.edu	65.613	47.935	26.898	100.178

Acknowledgements The authors would like to thank: Zachary M. Scott for code debugging; Dr. Sarah Neuwirth, Christian Melzer and Laura Promberger for European data; Dr. Ivan T. Ivanov at the Federal Reserve Board of Governors; E.G. White-Swift and Linda Merryman for proofreading; Gi Vania, Hassam Nasir, Dr. Christopher S. Simmons, Steven Kuhlo, and fellow members of the OpNeAR Lab for providing support and feedback for this paper.

Declarations

Funding Not Applicable.

Conflict of interest The authors declare that they have no conflict of interest.

Data availability Any data not presented directly in the paper is stored in an externally timestamped Git Repo (from gitlab.com) and is available upon reasonable request.

Code availability All source code is stored in an externally timestamped Git Repo (from gitlab.com) and is available upon reasonable request.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Abdul-Rahman, A., Hailes, S.: A distributed trust model. In: Proceedings of the 1997 Workshop on New Security Paradigms, NSPW '97, pp. 48–60. Association for Computing Machinery, New York, NY, USA (1998). <https://doi.org/10.1145/283699.283739>
2. Alimi, R., Penno, R., Yang, Y.R., Kiesel, S., Previdi, S., Roome, W., Shalunov, S., Woundy, R.: Application-layer traffic optimization (ALTO) protocol. RFC 7285, RFC Editor (2014). <https://tools.ietf.org/html/rfc7285>
3. Barisits, M., Beermann, T., Berghaus, F., Bockelman, B., Bogado, J., Cameron, D., Christidis, D., Ciangottini, D., Dimitrov, G., Elsing, M., Garonne, V., di Girolamo, A., Goossens, L., Guan, W., Guenther, J., Javurek, T., Kuhn, D., Lassnig, M., Lopez, F., Magini, N., Molfetas, A., Nairz, A., Ould-Saada, F., Prenner, S., Serfon, C., Stewart, G., Vaandering, E., Vasileva, P., Vigne, R., Wegner, T.: Rucio: Scientific data management. *Comput. Softw. Big Sci.* 3(1), 11 (2019). <https://doi.org/10.1007/s41781-019-0026-3>
4. Bartholomew, D.: Mariadb vs. mysql. Tech. rep., SkySQL AB (2012)
5. Bassi, A., Beck, M., Moore, T., Plank, J.S., Swany, M., Wolski, R., Fagg, G.: The internet backplane protocol: a study in resource sharing. *Future Gen. Comput. Syst.* 19(4), 551–561 (2003). Selected papers from the IEEE/ACM International Symposium on Cluster Computing and the Grid, Berlin-Brandenburg Academy of Sciences and Humanities, Berlin, Germany, 21–24 May 2002. [https://doi.org/10.1016/S0167-739X\(03\)00033-5](https://doi.org/10.1016/S0167-739X(03)00033-5). <https://www.sciencedirect.com/science/article/pii/S0167739X03000335>
6. Beck, M., Moore, T.: The internet2 distributed storage infrastructure project: an architecture for internet content channels. *Comput. Netw. ISDN Syst.* 30(22), 2141–2148 (1998)
7. Beck, M., Moore, T., Plank, J., Swany, M.: Logistical networking. In: Hariri, S., Lee, C.A., Raghavendra, C.S. (eds.) *Active Middleware Services*, pp. 141–154. Springer, US, Boston, MA (2000)
8. Berman, M., Chase, J.S., Landweber, L., Nakao, A., Ott, M., Raychaudhuri, D., Ricci, R., Seskar, I.: GENI: a federated testbed for innovative network experiments. *Comput. Netw.* 61, 5 – 23 (2014). Special issue on Future Internet Testbeds—Part I. <https://doi.org/10.1016/j.bjp.2013.12.037>. <https://www.sciencedirect.com/science/article/pii/S1389128613004507>
9. Braden, R.: Requirements for internet hosts—communication layers. RFC 1122, RFC Editor (1989). <https://www.rfc-editor.org/rfc/rfc1122.txt>
10. Brisco, T.: DNS support for load balancing. RFC 1794, RFC Editor (1995). <https://www.rfc-editor.org/rfc/rfc1794.txt>
11. Chervenak, A., Deelman, E., Foster, I., Guy, L., Hoschek, W., Iamnitchi, A., Kesselman, C., Kunszt, P., Ripeanu, M., Schwartzkopf, B., Stockinger, H., Stockinger, K., Tierney, B.: Giggie: a framework for constructing scalable replica location services. In: SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, pp. 58(2002). <https://doi.org/10.1109/SC.2002.10024>
12. Cisco Systems, Inc.: Sample Configuration for Authentication in OSPF (2005)
13. Cox, R., Muthitacharoen, A., Morris, R.T.: Serving DNS using a peer-to-peer lookup service. In: Druschel, P., Kaashoek, F., Rowstron, A. (eds.) *Peer-to-Peer Systems*, pp. 155–165. Springer, Berlin Heidelberg (2002)
14. Faltstrom, P., Kolkman, O.: The BLAKE2 cryptographic hash and message authentication code (MAC). RFC 7693, RFC Editor (2015). <https://www.rfc-editor.org/rfc/rfc7693.txt>
15. Faltstrom, P., Kolkman, O.: The uniform resource identifier (URI) DNS resource record. RFC 7553, RFC Editor (2015). <https://www.tools.ietf.org/html/rfc7553>
16. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol—HTTP/1.1. RFC 2616, RFC Editor (1999). <https://www.rfc-editor.org/rfc/rfc2616.txt>
17. Forster, F.: liboping(3) Linux User's Manual (2018). <http://nop-ing.cc/>
18. Gangan, S.: A review of man-in-the-middle attacks. CoRR [arxiv: 1504.02115](https://arxiv.org/abs/1504.02115) (2015)
19. Information Sciences Institute, University of Southern California: transmission control protocol. RFC 793, RFC Editor (1981). <https://www.rfc-editor.org/rfc/rfc793.txt>
20. Internet Systems Consortium, Inc.: dig(1) Linux User's Manual (2014)
21. Malkin, G.S.: RIP version 2. RFC 2453, RFC Editor (1998). <https://www.rfc-editor.org/rfc/rfc2453.txt>
22. McCallum, J.C.: Disk drive prices 1955+ (2021). <https://jcmitt.net/diskprice.htm>
23. McKenney, P.E.: Broadcast storms, nervous hosts, and load imbalances. *USENIX; Login* 13(5), 9–18 (1988)
24. Mockapetris, P.: Domain names—concepts and facilities. RFC 1034, RFC Editor (1987). <http://www.rfc-editor.org/rfc/rfc1034.txt>
25. Mockapetris, P.: Domain names—implementation and specification. RFC 1035, RFC Editor (1987). <http://www.rfc-editor.org/rfc/rfc1035.txt>

26. Mohyedinbonab, E., Jamshidi, M., Jin, Y.F.: A review on applications of graph theory in network analysis of biological processes. *Int. J. Intell. Comput. Med. Sci. Image Process.* **6**(1), 27–43 (2014). <https://doi.org/10.1080/1931308X.2014.938492>
27. Moy, J.: OSPF protocol analysis. RFC 1245, RFC Editor (1991). <https://www.rfc-editor.org/rfc/rfc1245.txt>
28. Moy, J.: OSPF version 2. RFC 2328, RFC Editor (1998). <https://www.rfc-editor.org/rfc/rfc2328.txt>
29. Pike, R., Kernighan, B.W.: The UNIX system: program design in the UNIX environment. AT&T Bell Lab. Techn. J. **63**(8), 1595–1605 (1984). <https://doi.org/10.1002/j.1538-7305.1984.tb00055.x>
30. ping(8) Linux User's Manual, bsd4.3 edn. (2018)
31. Ieee standard for information technology—portable operating system interface (POSIX(TM))—base specifications, issue 7—technical corrigendum 2. IEEE Std 1003.1-2008/Cor 2-2016 (Corrigendum to IEEE Std 1003.1-2008) The Open Group Technical Standard Base Specifications, Issue 7 pp. 1–310 (2016). <https://doi.org/10.1109/IEEESTD.2016.7542098>
32. pthreads(7) Linux User's Manual (2017)
33. Rekhter, Y., Li, T.: A border gateway protocol 4 (BGP-4). RFC 1654, RFC Editor (1995). <https://www.rfc-editor.org/rfc/rfc1654.txt>
34. Ritchie, D.M., Thompson, K.: The UNIX time-sharing system. *Commun. ACM* **17**(7), 365–375 (1974). <https://doi.org/10.1145/361011.361061>
35. Severance, C.: Introduction to networking, chap. 9, pp. 101–106. CreateSpace (2015)
36. Seymour, K., Yarkhan, A., Agrawal, S., Dongarra, J.: Netsolve: grid enabling scientific computing environments. In: *Grid Computing and New Frontiers of High Performance Processing*, volume 14 of *Advances in Parallel Computing*, p. 14. Elsevier, New York (2005)
37. staff, C.N.: Rogue domains revolt (1997). <https://www.cnet.com/news/rogue-domains-revolt/>
38. Stevens, W.R.: UNIX Network Programming, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ (1998)
39. Thomson, S., Huitema, C., Ksinant, V., Souissi, M.: DNS extensions to support IP version 6. RFC 3596, RFC Editor (2003). <https://www.rfc-editor.org/rfc/rfc3596.txt>
40. traceroute(8) Linux User's Manual (2006)
41. Vaillant, L., Scotoni, F., Savage, M.: Manual—introduction to monocpypher. <https://monocpypher.org/manual/>
42. Villars, R.L., Amaldas, M.: Rethinking your data retention strategy to better exploit the big data explosion. Dell Inc, Tech. rep. (2001)
43. Wasabi Technologies, Inc: Locations-2020. <https://wasabi.com/locations/>
44. Weber, S., Miotto, G., Almeida, J., Blanc, P., Dias, A., Malaguti, G., Manninen, H., Pfeifer, J., Ravat, S., Onnela, A., Mathot, S., Kirkbyz, J., Tomex, A., Amorim, A.: Data acquisition system of the cloud experiment at CERN. *IEEE Trans. Instrum. Meas.* **70**, 1–13 (2021). <https://doi.org/10.1109/TIM.2020.3023210>
45. Yovita, L.V., Syambas, N.R.: Caching on named data network: a survey and future research. *Int. J. Electr. Comput. Eng.* **8**(6), 4456–4466 (2018)
46. Zhang, L., Estrin, D., Burke, J., Jacobson, V., Thornton, J.D., Smetters, D.K., Zhang, B., Tsudik, G., kc claffy, Krioukov, D., Massey, D., Papadopoulos, C., Abdelzaher, T., Wang, L., Crowley, P., Yeh, E.: Named data networking (NDN) project. Tech. rep., NDN Project (2010). <http://www.named-data.net/techreports.html>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Joseph White-Swift is currently a Systems Engineer in the Office of Information Technology at the University of Texas at Dallas. He holds a B.S. in Electrical Engineering (2019) from University of Texas at Dallas. He is an Eagle Scout and in his day job works on HPC systems and Networking. His research interests include: Networking, Optical, and Protocols.

Andrea Fumagalli is a Professor of Electrical and Computer Engineering at the University of Texas at Dallas. He holds a Ph.D. in Electrical Engineering (1992) and a Laurea Degree in Electrical Engineering (1987), both from the Politecnico di Torino, Torino, Italy. From 1992 to 1998 he was an Assistant Professor of the Electronics Engineering Department at the Politecnico di Torino, Italy. He joined UT-Dallas as an Associate Professor of Electrical Engineering in August 1997 and was elevated to the rank of Professor in 2005. Dr. Fumagalli's research interests include aspects of wireless, optical, Internet of Things (IoT), and cloud networks, and related protocol design and performance evaluation. He has published about two hundred and fifty technical papers in peer-reviewed journals and conferences.